So, what's next?

Hi. I am (@zloirock), a full-time open-source developer. I don't like to write long posts, but it seems it is high time to do it. Initially, this post was supposed to be a post about the start of active development of the new major version of `core-js` and the roadmap (it was moved to the second half), however, due to recent events, it became a really long post about many different things... I'm fucking tired. Free open-source software is fundamentally broken. I could stop working on this silently, but I want to give open-source one last chance.

▶ 🔻 **Click to see how you can help** 🔻

If you or your company use `core-js` in one way or another and are interested in the quality of your supply chain, support the project:

- **Open Collective**
- **Patreon**
- **Boosty**
- **Bitcoin (bc1qlea7544qtsmj2rayg0lthvza9fau63ux0fstcz)**
- **Alipay**

Write me if you want to offer a good job on Web-standards and open-source.

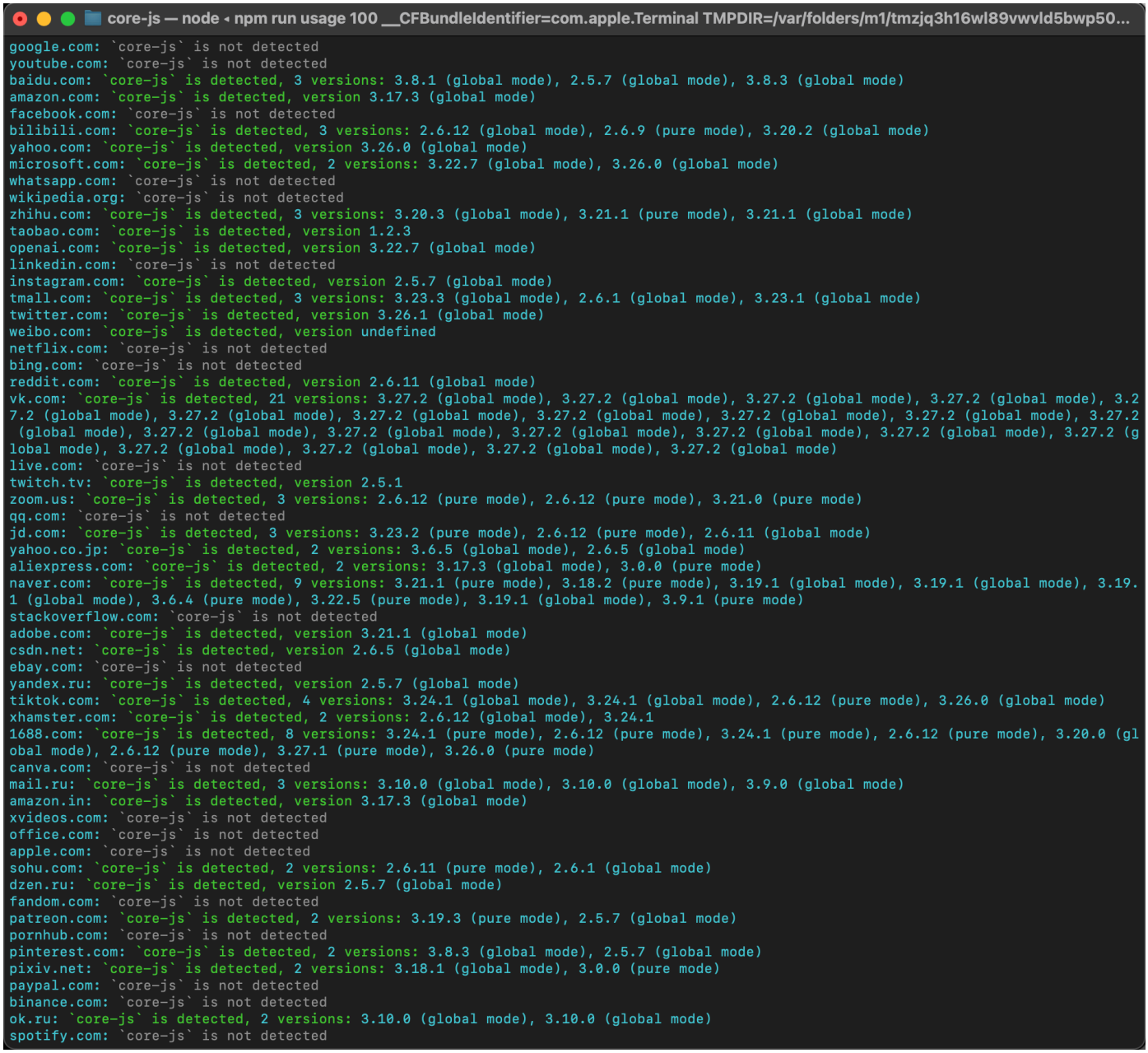
What is `core-js`?

- It is the most popular and the most universal polyfill of the JavaScript standard library, which provides support for the latest ECMAScript standard and proposals, from ancient ES5 features to bleeding edge features like iterator helpers, and web platform features closely related to ECMAScript, like `structuredClone`.
- It is the most complex and comprehensive polyfill project. At the time of publishing this post, `core-js` contains about half a thousand polyfill modules with different levels of complexity — from `Object.hasOwn` or `Array.prototype.at` to `URL`, `Promise` or `Symbol` — that are designed to work together. With a different architecture, each of them could be a separate package — however, it is not as convenient.
- It is maximally modular — you can easily (or even automatically) choose to load only the features you will be using. It can be used without polluting the global namespace (someone calls such a use case "ponyfill").
- It is designed for integration with tools and provides everything that's required for this — for example, `@babel/preset-env`, `@babel/transform-runtime`, and similar SWC features are based on `core-js`.

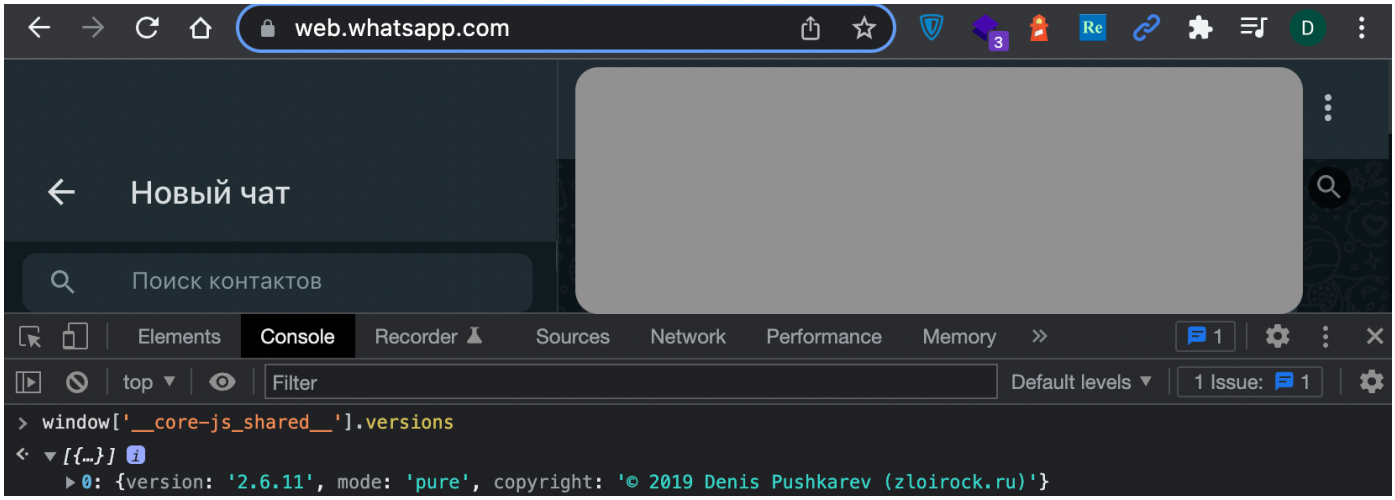
- It is one of the main reasons why developers could use modern ECMAScript features in their development process every day for many years, but most developers just don't know that they have this possibility because of `core-js` since they use `core-js` indirectly as it's provided by their transpilers / frameworks / intermediate packages like `babel-polyfill` / etc.
- It is not a framework or a library, whose usage requires the developer to know their API, periodically look at the documentation, or at least remember that he or she is using it. Even if developers use `core-js` directly — it's just some lines of import or some lines in the configuration (in most cases — with mistakes, since almost no one read the documentation), after that, they forget about `core-js` and just use features from web-standards provided by `core-js` — but sometimes this is the most of JS standard library that they use.

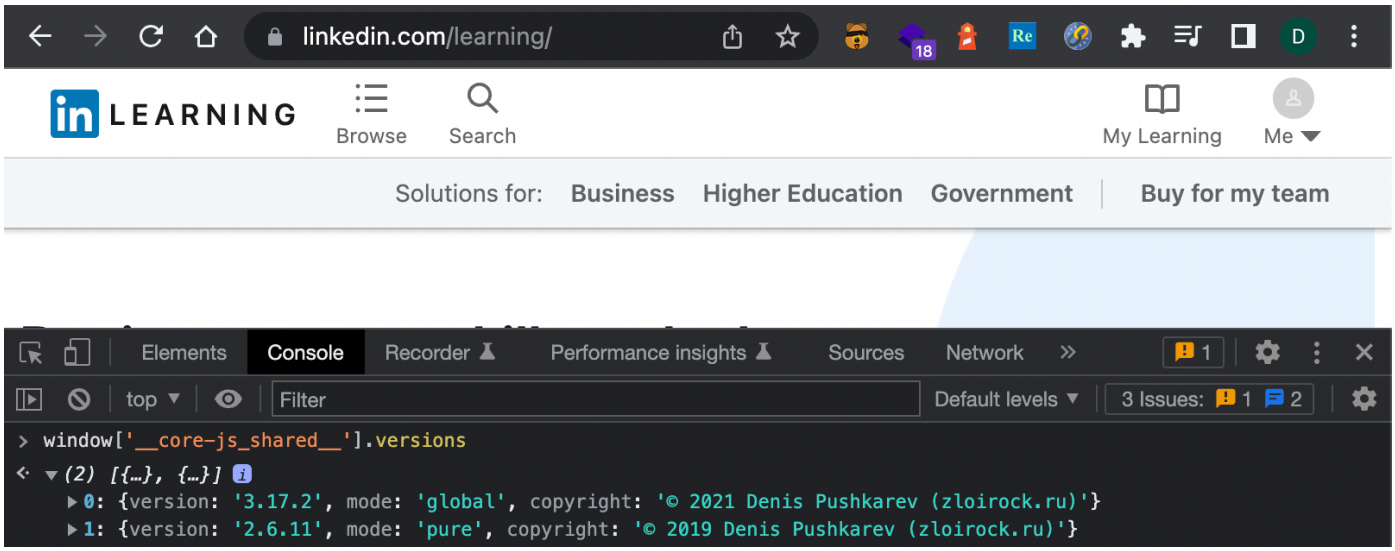
About 9 billion NPM downloads / 250 million NPM downloads for a month, 19 million dependent GitHub repositories (global ∪ pure) — big numbers, however, they do not show the real spread of `core-js`. Let's check it.

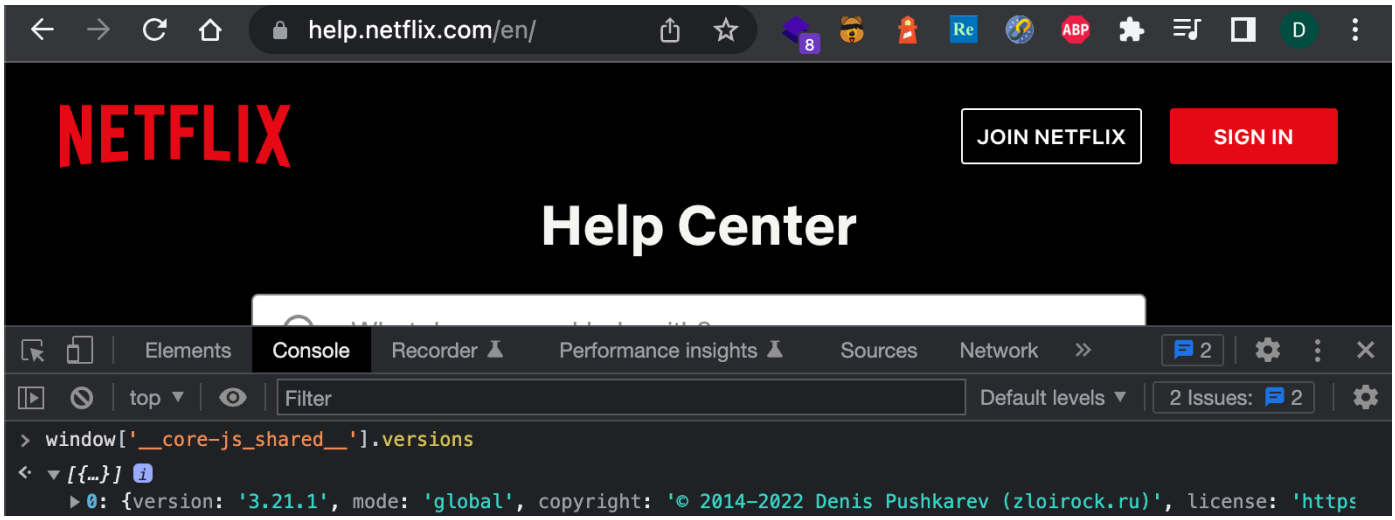
I wrote a simple script that checks the usage of `core-js` in the wild by the Alexa top websites list. We can detect obvious cases of `core-js` usage and used versions (only modern).

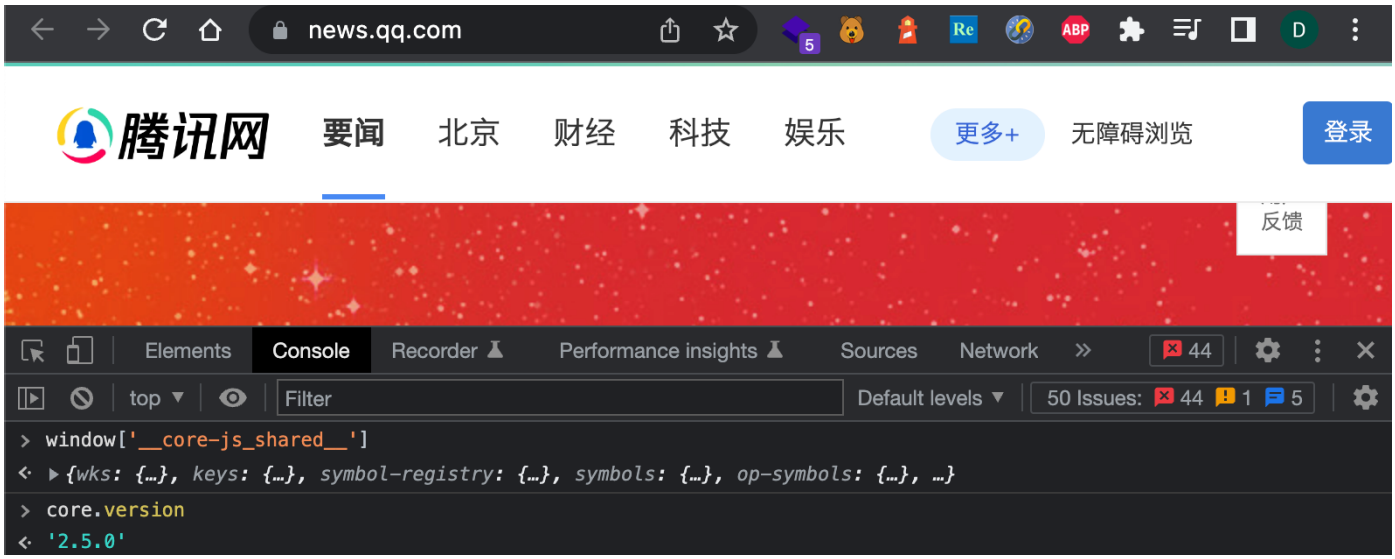


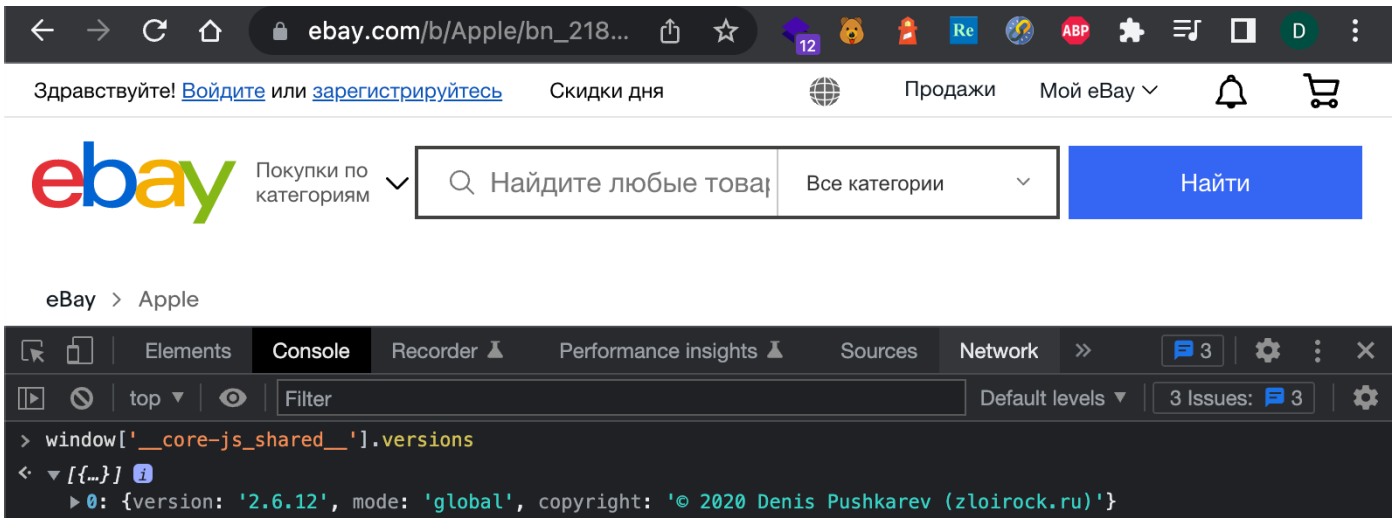
At this moment, this script running on the TOP 1000 websites **detects usage of `core-js` on 52% of tested websites**. Depending on the phase of the moon (the list, websites, etc. are not constants), results may vary by a few percent. However, it's just a naive detection on websites' home pages using a modern browser that loses many cases, **manual check shows that it's additional dozens of percent**. For example, let's leave the home pages of some websites from the screenshot above where `core-js` was **not** found by this script, without repeating for each company (at first — MS that's already on the screenshot) websites (be patient, after the series of screenshots the number of pictures will decrease):

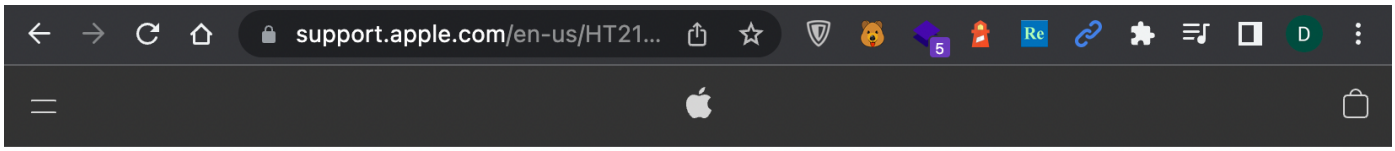




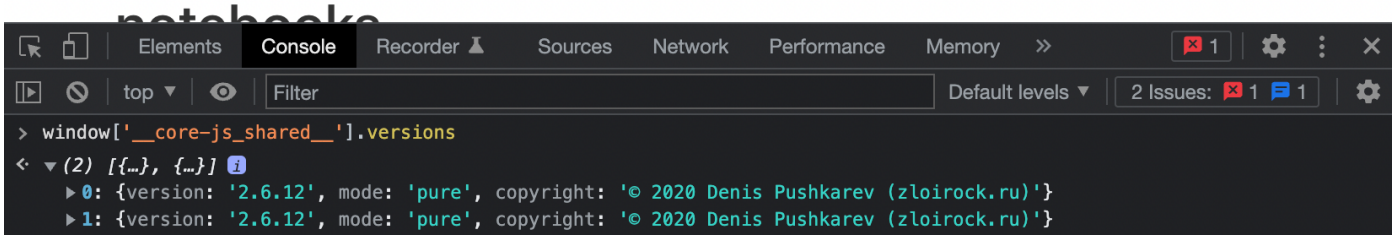

help.netflix.com/en/
NETFLIX
JOIN NETFLIX
SIGN IN
Help Center
Elements
Console
Recorder
Performance insights
Sources
Network
top
Filter
Default levels
2 Issues: 2
> window['__core-js_shared__'].versions
[{…}]
0: {version: '3.21.1', mode: 'global', copyright: '© 2014-2022 Denis Pushkarev (zloirock.ru)', license: 'https

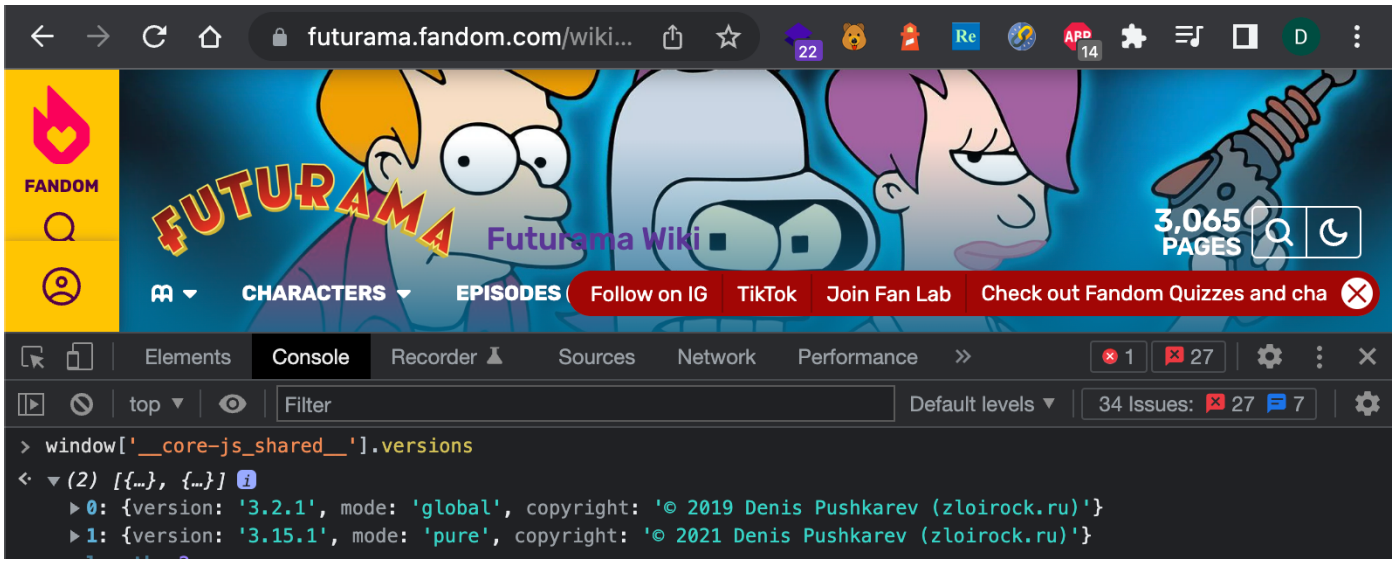

news.qq.com
腾讯网
要闻
北京
财经
科技
娱乐
更多+
无障碍浏览
登录
反馈
Elements
Console
Recorder
Performance insights
Sources
Network
44
top
Filter
Default levels
50 Issues: 44 1 5
> window['__core-js_shared__']
{wks: {…}, keys: {…}, symbol-registry: {…}, symbols: {…}, op-symbols: {…}, …}
> core.version
'2.5.0'

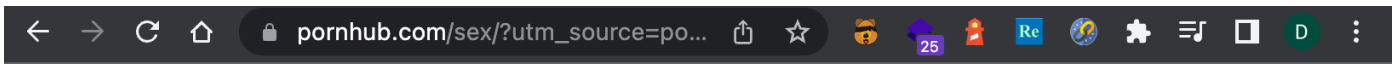

ebay.com/b/Apple/bn_218...
Здравствуйте! Войдите или зарегистрируйтесь
Скидки дня
Продажи
Мой eBay
ebay
Покупки по категориям
Найдите любые товар
Все категории
Найти
eBay > Apple
Elements
Console
Recorder
Performance insights
Sources
Network
3
top
Filter
Default levels
3 Issues: 3
> window['__core-js_shared__'].versions
[{…}]
0: {version: '2.6.12', mode: 'global', copyright: '© 2020 Denis Pushkarev (zloirock.ru)'}

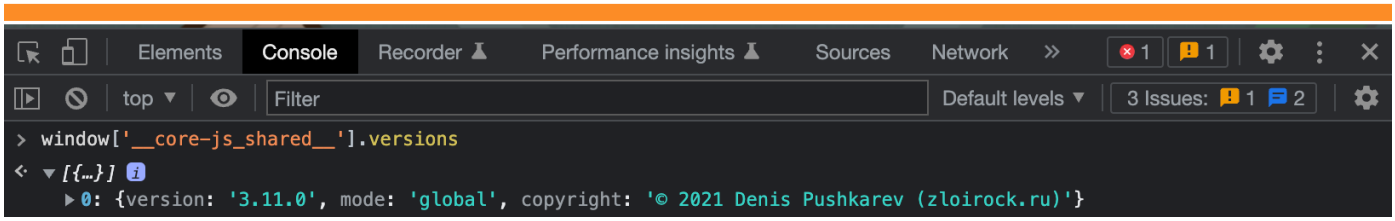


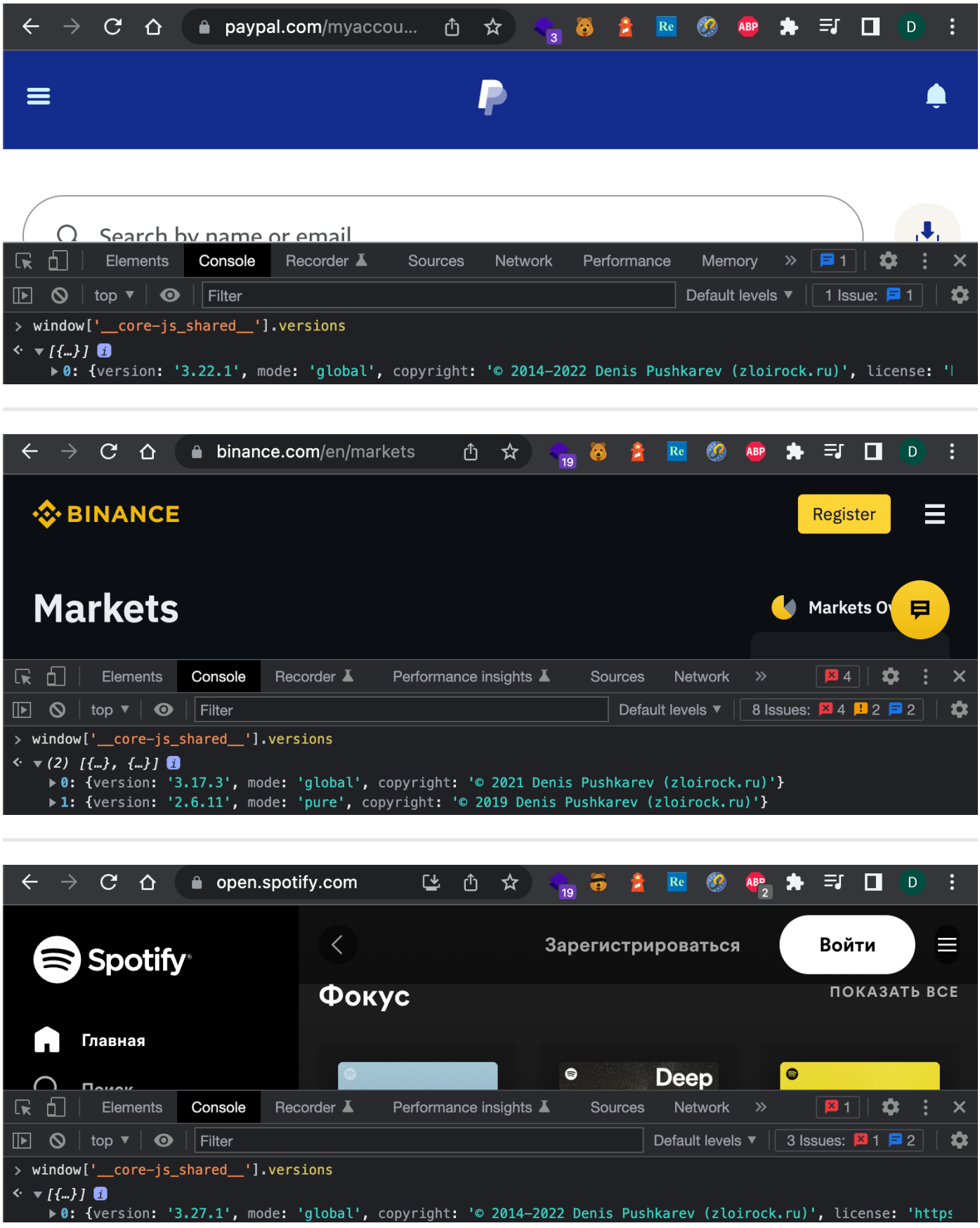
About battery health management in Mac notebooks











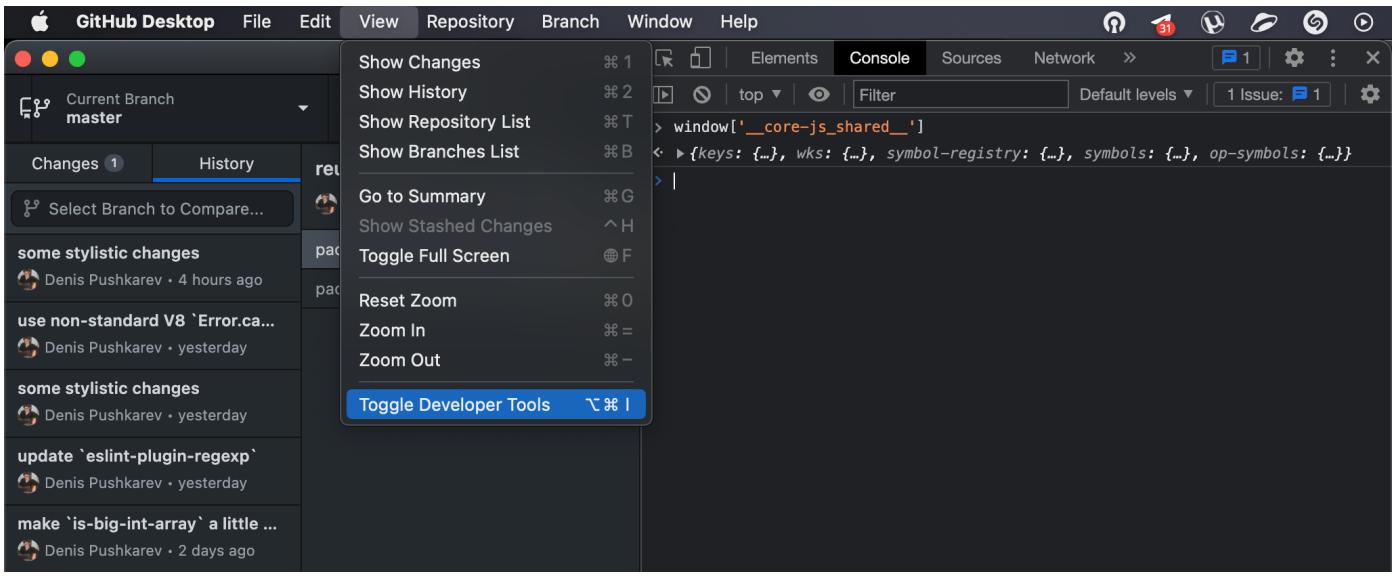
With such a manual check, you can find `core-js` on about 75-80% of the top 100 websites while the script found it on about 55-60%. On a larger sample the percentage, of course, decreases.

Wappalyzer allows to detect used technologies, including `core-js`, with a browser plugin and has previously shown interesting results, but now on their website, all the most popular technologies' public results are limited to only about 5 million positives. Statistics based on Wappalyzer results are available here and show `core-js` on 41% and 44% of 8 million mobile and 5 million desktop tested pages. Built With at this moment shows `core-js` on 54% of TOP 10000 sites (however, I'm not sure about the completeness of their detection and see the graph from another reality).

Anyway, we can say with confidence that **`core-js` is used by most of the popular websites**. Even if `core-js` is not used on the main site of any large corporation, it's definitely used in some of their projects.

What JS libraries are more widespread on websites? It's not React, Lodash, or any other most talked-about library or framework, I am pretty sure only about "good old" jQuery.

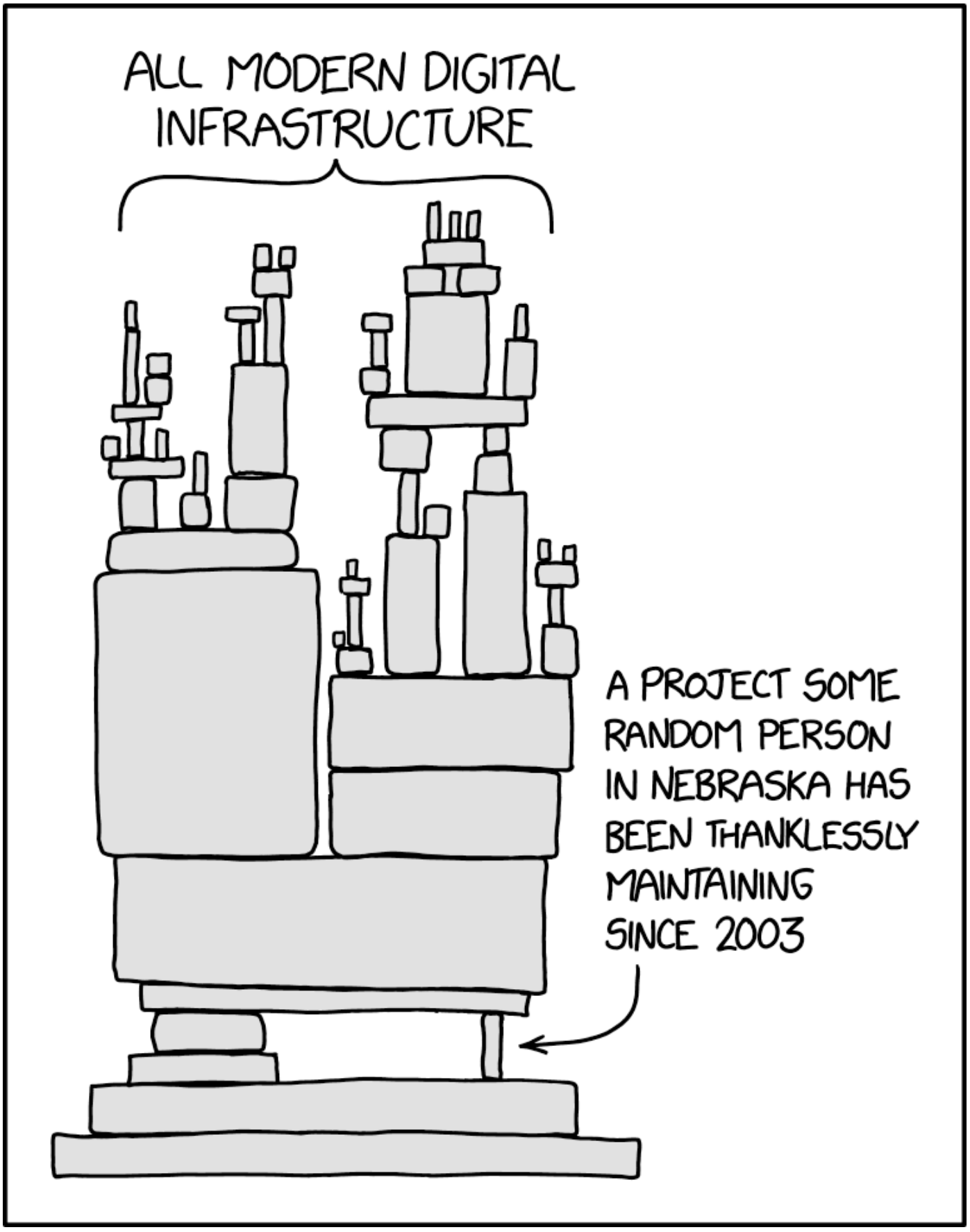
And `core-js` is not only about a website's frontend — it's used almost everywhere where JavaScript is used — but I think that's more than enough statistics.



However, for the above reasons, **almost no one remembers that he or she uses `core-js`**.

Why am I posting this? No, not to show how cool I am, but to show how bad everything is. Read on.

Let's start the next part with one popular `xkcd` picture


ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

Beginning

I switched my development stack to full-stack JavaScript in 2012. It was a time when JavaScript still was too raw — IE still was more popular than anything else, ES3 era browsers still occupied a significant part of the web, the latest NodeJS version was 0.7 — it was just starting its way. JavaScript still was not adapted for writing of serious applications and developers solved the lack of required language syntax sugar with compilers from languages like CoffeeScript and the lack of proper standard library with libraries like Underscore. However, it wasn't a standard — over time, these languages and libraries became obsolete together with the projects that used them. So, I took all news of the upcoming ECMAScript ~~Harmony~~ 6 standard with great hope.

Given the prevalence of old JavaScript engines and the fact that users were in no hurry and often did not have the opportunity to abandon them, even in the case of quick and problem-free adoption of the new ECMAScript standard, the ability to use it only through JavaScript engines was postponed for many and many years. But it was possible to try to get support features from this standard using some tools. Transpilers (this word was not as popular as it is now) should have to solve the problem with the syntax, and polyfills — with the standard library. However, at that time the necessary toolkit was only just beginning to emerge.

It was a time when ECMAScript transpilers started to become popular and develop actively. However, at the same time, polyfills have barely evolved according to users' and real-life projects' needs. They were not modular. They could not be used without global namespace pollution — so they were not suitable for libraries. They weren't a single complex — it was required to use multiple different polyfill libraries from different authors and somehow make them work together — but in some cases, it was almost impossible. Too many necessary fundamental language features were just missing.

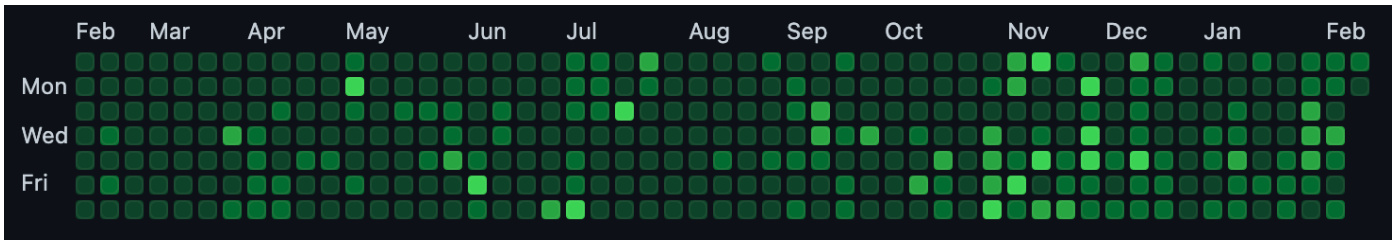
To fix these problems, at the end of 2012, initially for my own projects, I started to work on a project that was later called `core-js`. I wanted to make the life of all JS developers easier and in November 2014, I published `core-js` as an open-source project. *Maybe it was the biggest mistake in my life.*

Since I was not the only one who faced these issues, after a few months, `core-js` has already become the de facto standard of polyfill for JavaScript standard library features. `core-js` was integrated into Babel (`6to5` at that moment) which appeared a couple of months before `core-js` was published — some of the aforementioned issues were critical for that project too. `core-js` began to be distributed as `6to5/polyfill`; and after rebranding as `babel-polyfill`. After a few months of collaboration, a tool has appeared, which became `babel-runtime` after rebranding and evolution. A few months later `core-js` was integrated into the key frameworks.

Ensuring compatibility for the whole Web

I didn't promote myself or the project. *This is the second mistake.* `core-js` didn't have a website or social media accounts, only GitHub. I did not show up at conferences to talk about it. I wrote almost no posts about it. I was just making a really useful and wanted part of the modern development stack, and I was happy about that. I gave developers a chance to use the most modern and really necessary JavaScript features without waiting for years until they are implemented in all required engines, without thinking about compatibility and bugs — and they started to use it. The spread of the project had grown exponentially — very soon it was already used on dozens of percent of popular websites.

However, it was just the start of the required work. Many years of hard work followed. Almost every day I spent some hours on `core-js` and maintenance of related projects (mainly Babel and `compat-table`).



`core-js` is not a several lines library that you can write and forget about it. Unlike the vast majority of libraries, it's bound to the state of the Web. It should react to any change of JavaScript standards or proposals, to any new JS engine release, to any detection of a bug in JS engines, etc. ECMAScript ~~6~~ 2015 was followed by new proposals, new versions of ECMAScript, new non-ECMAScript web standards, new engines and tools, etc. The evolution, the improvement of the project, and the adaptation to the current state of the Web have never stopped — and almost all of this work remains not visible to the average user.

The scale of required work was constantly growing.

I tried to find other maintainers or at least constant contributors for `core-js` in different ways for a long time, but all attempts have failed. Almost every JS developer used `core-js` indirectly and knew, for example, `babel-polyfill`, `babel-runtime`, or that their framework polyfilled all required features, but almost no one knew `core-js`. In some posts about polyfilling where `core-js` was mentioned, it was called "a small library". It was not a trendy and widely discussed project, so why help maintain it if it works great anyway? Over time, I lost hope for it, but I felt a responsibility to the community, so I was forced to continue working alone.

After a few years combining full-time work and FOSS became almost impossible — no one wanted to pay money for the working time devoted to FOSS, non-working hours were not enough, and sometimes `core-js` required complete immersion for weeks. I thought that proper polyfilling is required for the community and money was not my priority.

I left a high-paying job and did not accept some very good options because in those positions I would not have had the opportunity to devote enough time to work on open-source. I started to work on open-source full-time. No one paid me for it. I hoped sooner or later to find a job where I could fully dedicate myself to open-source and web standards. Periodically, I earned the money required for living and work on FOSS, on short-term contracts. I returned to Russia, where it was possible to have a decent standard of living with relatively little money. *One more mistake — as you will see below, money matters.*

Until April 2019, for about one and a half years as a whole and about a half-year full-time without distraction of any other work, I worked on the `core-js@3` with a fundamental improvement of polyfilling-related Babel tools, the foundation of the toolkit generation that now is used almost everywhere.

Accident

Shit happened 3 weeks after the `core-js@3` release. One April night, at 3 AM, I was driving home. Two deadly drunk 18-years-old girls in dark clothes decided somehow *to crawl* across a poorly lit highway — one of them laid down on the road, another sat down and dragged the first, but not from the road — directly under my wheels. That's what the witnesses said. I had absolutely no chance to see them. One more witness said that before the accident they were just jokingly fighting on the road. Nothing unusual, it's Russia. One of them died and another girl went to the hospital. However, even in this case, according to Russian arbitrage practice, if the driver is not a son of a deputy or someone like that, he would almost always be

found guilty — he has to see and anticipate everything, and a pedestrian owes nothing to anyone. I could end up in prison for a long time, IIRC later the prosecutor requested 7 years.

The only way not to end up in prison was reconciliation with "victims" — a standard practice after such accidents — and a good lawyer. Within a few weeks after the accident, I received financial claims totaling about 80 thousand dollars at the exchange rate at that time from "victims'" relatives. A significant amount of money was also needed for a lawyer.

Maybe it's not an inconceivable amount of money for a good software engineer, but, as I wrote just above, I worked full time on the `core-js@3` release for a long time. Of course, no one paid me for this work, and I completely exhausted all my financial reserves, so, sure, I didn't have that kind of money and I didn't have a chance to find the money required from available sources. The time I had was running out.

Fundraising

By that time `core-js` already was used almost as widely as it is now. As I wrote above, I looked for contributors for `core-js` for a long time without any success. However, `core-js` is a project that should be actively maintained and it can't stay just frozen. My long-term imprisonment would have caused problems not only for me — but it would have also been the death of `core-js` and a problem for everyone who had been using it — for half of the Web. The notorious bus factor.

Some months before that, I started raising funds to support the `core-js` development (mainly it was posted in READMEs on GitHub and NPM). The result was... $57 / month. Fair pay for full-time work on ensuring compatibility for the whole web 😂

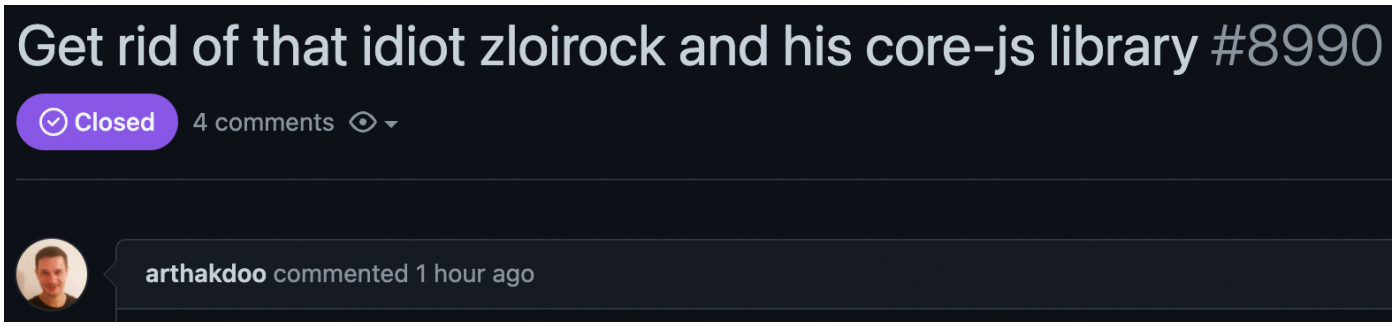
I decided to do a little experiment — to ask for help from the `core-js` users — those who will suffer if `core-js` will be left without maintenance. I added a message in `core-js` installation:

```
Thank you for using core-js ( https://github.com/zloirock/core-js ) for polyfilling JavaScript standard library!

The project needs your help! Please consider supporting of core-js on Open Collective or Patreon:
> https://opencollective.com/core-js
> https://www.patreon.com/zloirock

Also, the author of core-js ( https://github.com/zloirock ) is looking for a good job -)
```

I understood that I'd have hardly gotten all the required money from donations, however, every dollar mattered. I added a job search message to get a chance to earn the other part. I was thinking that a few lines in the NPM installation log asking to help, which can be hidden if needed, is an acceptable price for using `core-js`. The original plan was to delete this message in a few weeks, but everything went against the plan. How wrong I was about people...

Hate

Of course, I expected that someone would not like to see a request for help in their console, but the continuous stream of hate that I began to receive went through the roof. It was hundreds of messages, posts, and comments every day. All of it can be reduced to something like:



This is far from the funniest thing I've seen — if I wanted to, I could collect a huge selection of statements in the style collected here — but why? I already have enough negativity in my life.

Developers love to use free open-source software — it's free and works great, they are not interested in the fact that many and many thousands of hours of development, and real people with their own problems and needs are behind it. They consider any mention of this as an invasion of their personal space or even a personal affront. For them, these are just gears that should automatically change without any noise and their participation.

So, thousands of developers attacked me with insults and claimed that I have no right to ask them for any kind of help. My request for help offended them so much that they began to demand restricting my access to the repository and packages and move them to someone else like it was done with `left-pad`. Almost none of them understood what `core-js` does, the scale of the project, and, of course, nobody wanted to maintain it — it should be done by "the community", someone else. Seeing all this hatred, in order to not be led by the haters, I did not delete the help-asking message, which was initially planned to be there only for a couple of weeks, just out of principle.

What about companies which `core-js` helped and is helping to make big money? It's almost every big company. Let's rephrase this old tweet:

> Company: "We'd like to use SQL Server Enterprise"
>
> MS: "That'll be a quarter million dollars + $20K/month"
>
> Company: "Ok!"
>
> ...
>
> Company: "We'd like to use core-js"
>
> core-js: "Ok! npm i core-js"
>
> Company: "Cool"
>
> core-js: "Would you like to help contribute financially?"
>
> Company: "lol no"

A few months later, tired of user complaints, NPM presented `npm fund` — it was not a solution for the problem, it was just a way to get rid of those complaints. How often did you call `npm fund`? How often did you donate to someone who you saw in `npm fund`? Who did you see and support at first — `core-js` or someone who maintains a dozen of one-line libraries dependent on each other? It also provided NPM with a perfect justification for the future step (read below).

Within 9 months many thousands of developers, including developers of projects fundamentally dependent on `core-js`, knew about the situation — but no one offered to maintain `core-js`. Within many months I talked with maintainers of some significant projects dependent on `core-js`, but without any success — they didn't have the necessary time resources. So I was forced to ask some of my friends who were not related to FOSS community (at first **@slowcheetah**, thanks him for his help) to cover for me and at least try to fix significant issues until I get free.

Few users and small companies supported the `core-js` — and I am very grateful to them. However, the amount of money raised in 9 months was only about 1/4 of the money that should have been collected within a couple of weeks to change something.

During the same time, despite everything, the number of `core-js` downloads per day almost doubled.

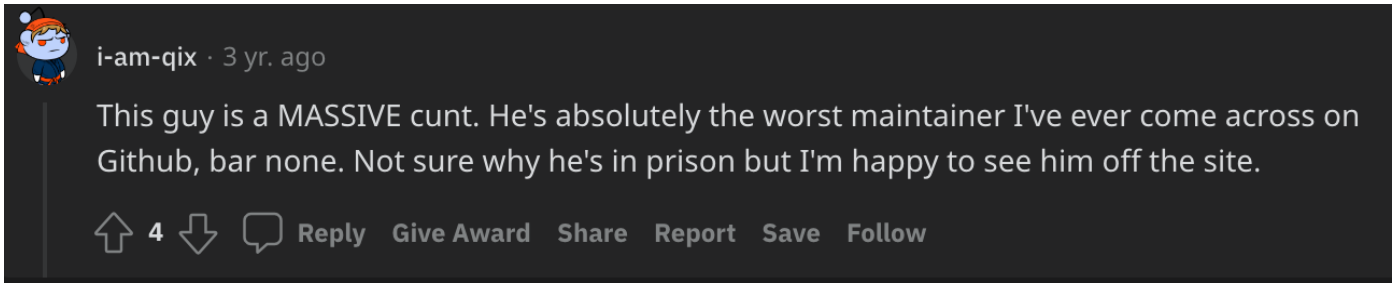
In January 2020 I ended up in prison.

Release

I don't wanna say many words about prison and I have no great desire remembering this. It was slave labor at a chemical factory where my health was significantly ruined and where I 24/7 had a great time in a company of drug dealers, thieves, and killers (from other regimes), without access to the Internet and computers.

After about 10 months, I was released early.

I saw dozens of articles, hundreds of posts, and thousands of comments the essence of many of which can be expressed by this:



What do you think I did? *Of course, I made the same mistake.* I saw some people who supported the development of `core-js`, many issues, questions, and messages — sure, not as many as angry comments. `core-js` became even more popular and was already used by almost the same percentage of websites as it is now.

Ensuring compatibility for the whole Web again

I returned to `core-js` maintenance like it was before. Moreover, I completely stopped being distracted by contracts and any other work in favor of working on `core-js`. `core-js` had some money on funding platforms — not so much, many times less than I received before starting work on `core-js` full-time — but for me alone it was enough to live on. A kind of down-shifting, full-time Open-Source to make the world better... I didn't think about the tens of thousands of dollars in lawsuits left over from the accident. I didn't think about my future. I thought about a better future for the Web. And, of course, I was hoping that some company would offer me a position with the opportunity to work on web standards and would sponsor my

work on polyfills and FOSS.

A lot has been accomplished over the next two years — in terms of work, almost as much as in the previous 8 years. This is still `core-js@3` — but much better. However, the changelog and even the previous diff reflect only a small percentage of the work done. Almost all of this work remains in the shadows, not visible to the average user.

It is the fundamental work with standards and proposals. As a side effect of this work, taking into account the hard work that was done and changes after my feedback and suggestions, I consider many of the ECMAScript proposals — that have become part of the language — are my achievements as much as they are achievements of their champions. It is the work with engines and their bug trackers in searching for bugs. It is the constant automatic and (too often) manual testing in many hundreds of environments, many thousands of environments / builds / test suites combinations to ensure proper operation of the standard library everywhere and to collect compat data. From a raw prototype, made in a couple of days, `core-js` compat data became an exhaustive data set with proper external and internal tooling. It is the design and prototyping of many features that are yet to appear in the project. And also much, much more.

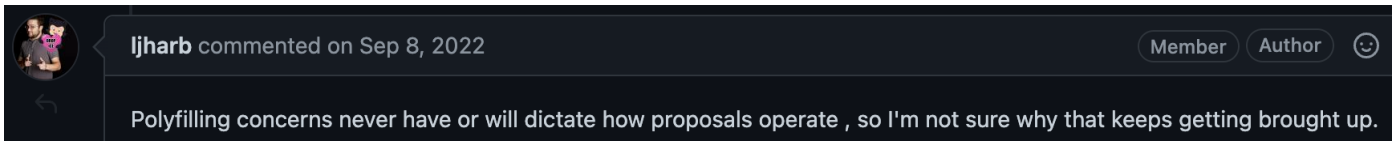
As you can see above, `core-js` is present in most of the popular websites, provides an almost complete JavaScript standard library, and fixes improper implementations. The number of web page openings with `core-js` is greater than the number of web page openings in Safari and Firefox. Thus, from a certain point of view, `core-js` can be called one of the most popular JavaScript runtimes.

When working on `core-js`, I am the first implementer of almost all modern and future JavaScript standard library features, almost all of them have my feedback and they have been fixed according to it. `core-js` is the best playground for experimentation with ECMAScript proposals. In too many cases, proposals receive feedback from other users after they play with experimental `core-js` implementations of these proposals.

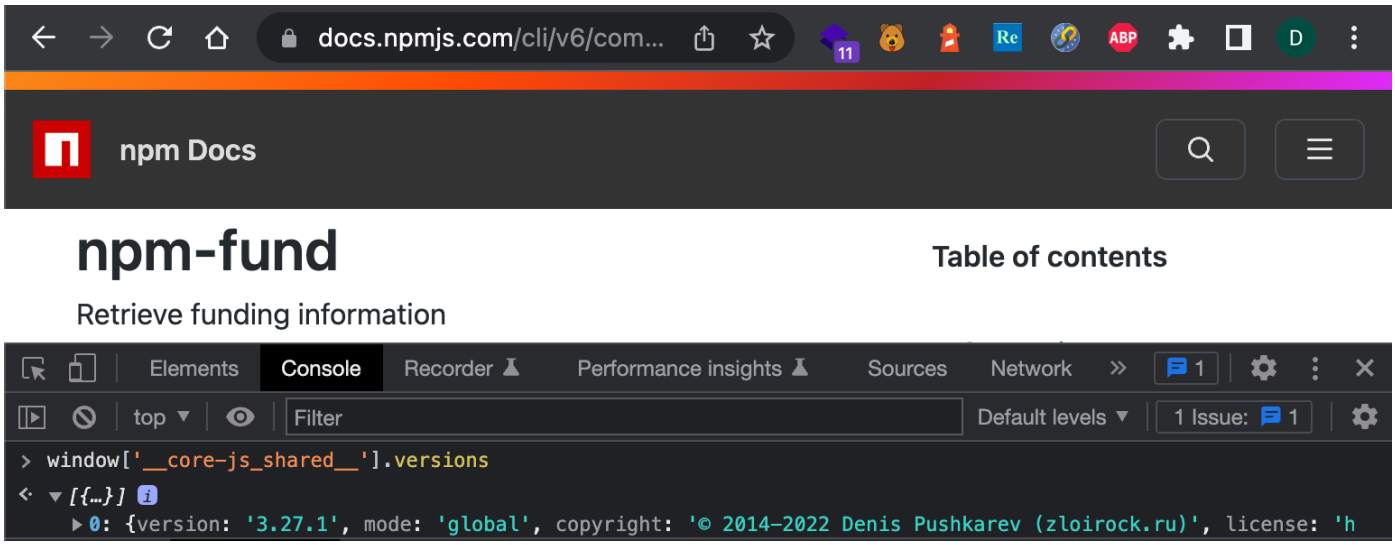
The best way forward for JavaScript would be for TC39 and `core-js` to work together on the future of JavaScript. For example, TC39 invites members of projects like Babel and others as experts. Except `core-js`. Instead, too often, I see the ignoring of my or `core-js`'s issues or even creation of roadblocks by TC39 members; and they don't even hide it:

> <shu> the real difficulty there is i now refuse to engage with the author of core-js

---



After a while, "support" came from NPM. In `npm@7`, which was released at the end of 2020, as a logical continuation of `npm fund`, the console output was disabled in post-install scripts. The result was expectable, because people stopped seeing the funding request and almost no one uses `npm fund`, the number of `core-js` backers began to decline. An excellent support for the project from those, who not only earn by distributing my work, but also use it themselves :-)



In addition, another factor came into play again. Higher quality — less support. Is the library well-maintained? There are practically almost no open bug reports, and when they happen, they are fixed almost instantly? Does the library already give us almost everything we want? Yes? So why should we support the maintenance of the library? The price at which this is done for the maintainers is not on the surface — for most developers and companies, it's still just "a small library". Many of those, who backed `core-js` before, stopped doing it.

The `core-js` code contains my copyright. As you can see at the top of this post, it's present in about half of all websites. Regularly someone finds it in the source code of harmful sites / applications — but they don't know what `core-js` is and their tech level is not good enough even to find it out. When this happens, the police will call and threaten me, and someone even tried to blackmail me. Sometimes it's not funny at all.

I have been contacted several times by American and Canadian journalists who discovered `core-js` on American news and government websites. They were very disappointed that I was not an evil Russian hacker who meddles in American elections.

The endless stream of hatred decreased slightly over time but continues. However, most of it moved from something like GitHub issues or Twitter threads to my mail or IM. Today, one developer wrote me a message. He called me a parasite on the body of the developer community that makes a lot of money spamming and doing nothing useful. He called me the same murderer as Hans Reiser, but who bought the judge and escaped unpunished. He wished death for me and all my relatives. And there is nothing unusual here, I get several of such messages a month. Last year, one more thing was added that I am a "Russian fascist".

Some words about the war

Open-source should be out of politics.

I don't want to choose between two kinds of evil. I will not comment on this in more detail, since there are people close to me on both sides of the border who may suffer because of this.

Let me remind you what I wrote about above: I returned to Russia because it was a place where it was possible to have a decent standard of living for relatively little money and to concentrate on FOSS instead of making money. Now I cannot leave Russia, because after the accident I have outstanding lawsuits in the amount of tens of thousands of dollars and I am forbidden to leave the country until they are paid off.

What do you think, how much money does `core-js` receive each month?

When I started to maintain `core-js` full-time, without being distracted by contracts and any other work, **it was about $2500 per month — it was about 4-5 times less than I usually had on full-time contracts**. Remember, a kind of down-shifting, to make the Web better. Temporarily. Reduce issues and bugs to zero, make the highest quality product, which is used by almost everyone... and the project will be sufficiently supported, right? Right?

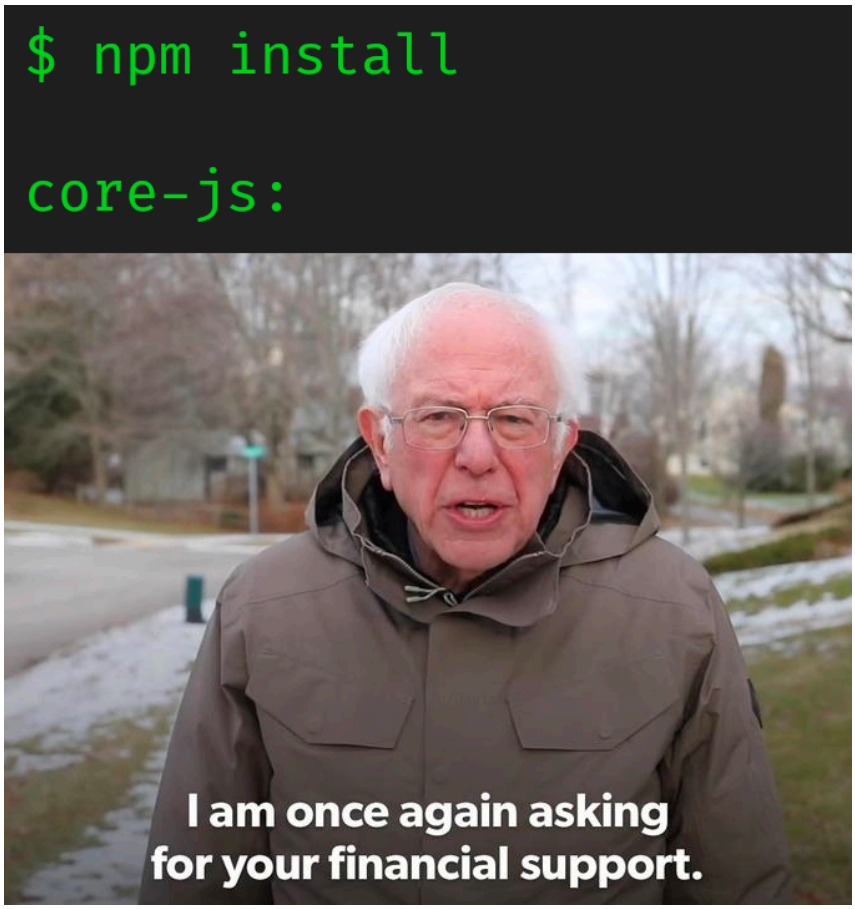
After a few months, the reoccurring monthly income **decreased to about $1700** *(at least that's what I thought)*, $1000 via Tidelift, $600 via Open Collective, and $100 via Patreon. In addition to the reoccurring monthly, one-time donations came periodically (on average it was maybe $100 per month).

Crypto? Adding a crypto wallet for donations was a very popular request. However, for all the time, only 2 transfers for a total amount of about $200 have been received on crypto wallets, the last one was more than a year ago. GitHub sponsors? It's not available in Russia and never was. PayPal? It's banned for Russians. When it was available, `core-js` received about $60 in all that time. Grants? I applied for a lot of grants — all applications were ignored.

The main part, $400 per month, of those donations, `core-js` received from... Bower, another FOSS community. I am also very grateful to all other sponsors: because of your donations, I'm still working on this project.

However, in this list there is not a single big corporation or at least a company from the top 1000 website list. Let's be honest, there are mainly individuals, and only a few small companies on the current list of backers and they pay a few dollars a month.

If someone says that they don't know that `core-js` requires funding... Come on, I regularly see memes like this:



A year ago, Tidelift stopped sending me money. They said that because of the political situation, the Hyperwallet, that they used, is no longer available to Russians (but it was available to me till last month when I tried to update some personal data), and for safety, they will store my money on their side. Over the previous couple of months, I tried to get this money to a bank or a Hyperwallet account, but only received replies that they will try to do something (*sounds great, doesn't it?*). Since the end of the last year, they have just stopped responding to emails. And now, I've got this:

Jeremy Katz jeremy@tidelift.com February 8 at 1:35
Me and Joshua Simmons ›

Reply all Forward Delete More

Hi Denis -

Apologies for the delay. Unfortunately, we cannot pay you if your account in Hyperwallet is frozen, so we are terminating your lifter agreement effective immediately. If you are able to unfreeze your Hyperwallet account, please let us know and we can re-establish the relationship.

Thanks

Jeremy

Denis Pushkarev zloirock@zloirock.ru february 5 at 12:21
Guys?

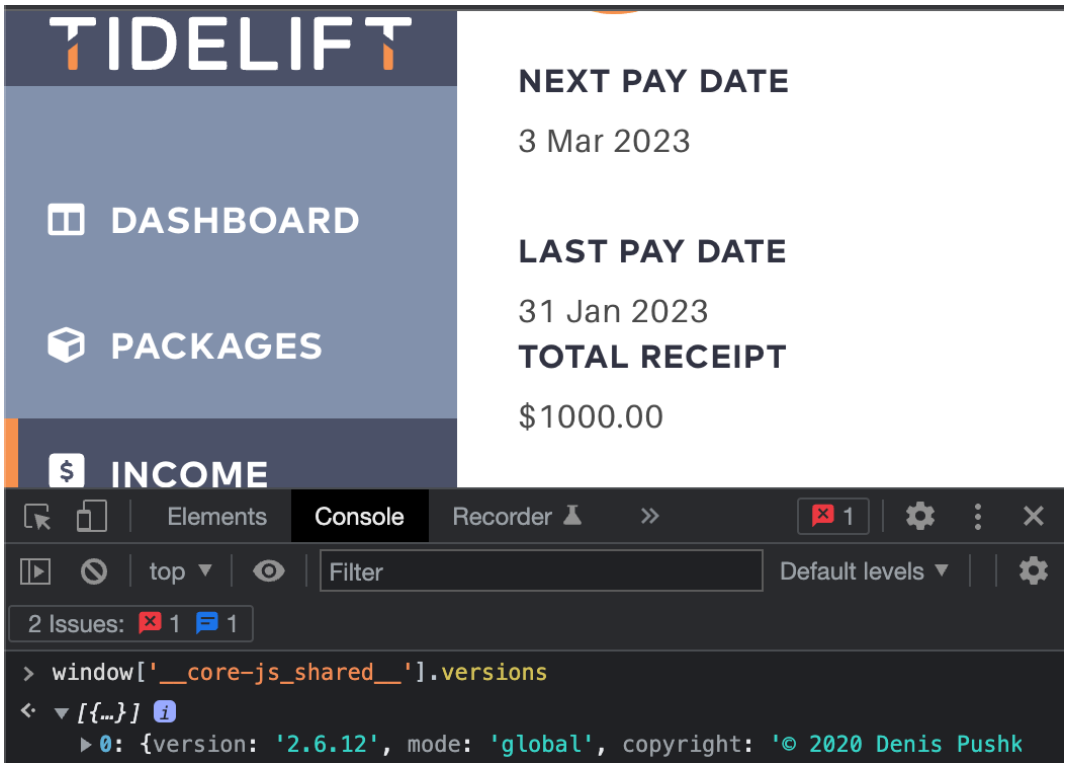
Denis Pushkarev zloirock@zloirock.ru february 2 at 0:10
Hey guys?

Denis Pushkarev zloirock@zloirock.ru january 29 at 23:34
Hey, a month has passed and my situation is getting critical.

Jeremy Katz jeremy@tidelift.com 29 december 2022 at 23:08
Let me see what I can run down... we're running on a bit of a skeleton crew this week in between the holidays

- Jeremy

In such an amusing way, I found out that I will not receive the money for the previous year, and this year I worked not for $1800, but for $800 a month. There were, of course, no replies to subsequent emails. However, their site indicated that I received and still receive money through them.



I wonder how the companies that support their dependencies chain through them will react to such a scam.

On the same day, on OpenCollective I saw that the reoccurring monthly was reduced from about $600 to about $300. Apparently, the financial reserves of `Bower` have come to an end. This means that **for this month I'll get about $400 total**.

In the previous months, I measured how much time it takes to work on `core-js`. It turned out about... **250 hours a month** — significantly more than a full day without any days off, which makes it impossible to have a "real" (as many say) full-time work or work for any significant contracts. $400 for 250 hours... It will be less than **$2 per hour of work, for the year before a little more: $4 per hour**. Yes, in some months, I did spend less time working on the project, but it does not change much.

This is the current price for ensuring compatibility for the whole Web. And no insurance or social security.

Awesome earning growth and career, right?

I think you understand well how much senior software engineers in key IT companies get paid. I received a lot of comparable offers, however, they are not compatible with the proper work on `core-js`.

Among the regular threats, accusations, demands, and insults, I often get something like "Stop begging and go to work, idler. Remove your beggarly messages immediately — I don't wanna see them." The funny thing is that at least some of these people get over $300,000 a year (which I know for sure because I talk to their colleagues), and (because of the nature of their work) `core-js` saves them many hours of work each month.

Everything changes

When I started working on `core-js`, I was alone. Now I have a family. A little over a year ago, I became a father of my son. Now I have to provide him with a decent standard of living.

I have a wife, and sometimes she wants some new shoes or a bag, a new iPhone or an Apple Watch. My parents are already at the age that I need to significantly support them.

I think it is obvious that it is impossible to properly support a family with the money that I have or had from `core-js` maintenance. Financial reserves that I used, have finally come to the end.

More and more often I hear reproaches like: "Give up your Open-Source, this is pampering. Go back to a normal job. `%USERNAME%` has been working as a programmer for just a year. He understands almost nothing about it, works a couple of hours a day, and already earns many times more than you do."

NO MORE

I'm damn tired. I love working on open-source and `core-js`. But who or what am I doing this for? Let's summarize the above.

- I have been ensuring zero compatibility issues and providing bleeding edge features of the web platform for most of the Web since 2014; and I've been working on it for most of the time for money, that now will not even be enough for food.
- Rather than any gratitude, all I see is the huge hatred from developers whose life I simplify.
- Companies that save and earn many millions of dollars on `core-js` usage just ignore `core-js` funding requests.

- Even in a critical situation, in response to a request for help, instead of help, most of them preferred to ignore or hate.
- Instead of working together with standards' and browsers' developers on a better future for JavaScript, I'm forced to struggle with roadblocks that they make.

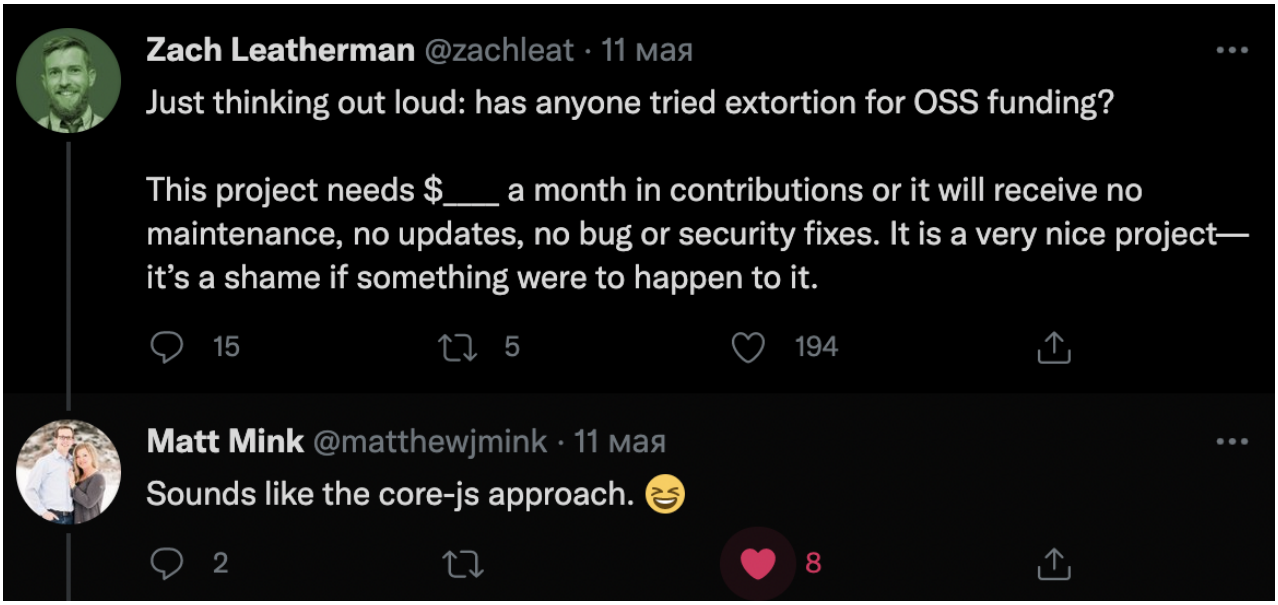
I don't care about the haters. Otherwise, I would have left open-source a long time ago.

I can tolerate the lack of normal interaction with the standards developers. First of all, this means future problems for users and, when the Web will be broken, for standards developers themselves.

However, money matters. I've had enough of sponsoring corporations at the expense of my and my family's well-being. I should be able to ensure a bright future for my family, for my son.

The work on `core-js` occupies almost all of my time, more than a full working day. This work ensures the proper functioning of the most of the popular websites and this work should be paid properly. I'm not going to keep working for free or for $2 per hour. I'm willing to continue working on a project for at least $80 an hour. This is the rate that have, for example, `eslint` team members per hour. And, if the work on open-source requires it, I'm ready to pay off my lawsuits and leave Russia — however, it's not cheap.

Regularly I see comments like this:



Ok guys, if you want it — let's use such an approach.

Depending on your feedback, `core-js` will soon follow one of the following ways:

- **Appropriate financial backing**

  I hope that, at least after reading this post, corporations, small companies, and developers will finally think about the sustainability of their development stack and will properly back `core-js` development. In this case, `core-js` will be appropriately maintained and I'll be able to focus on adding a new level of functionality.

The scale of the necessary work goes through the roof, a single me is no longer enough — I can't work more physically. Some work, for example, improving test coverage or documentation, is simple enough and takes a lot of time, but it's not the kind of work that volunteers want to do — I don't remember any PRs with improvements for test coverage of existent features. So it makes sense to attract at least one or two developers (at least students, better — higher level) on a paid basis.

Taking into account the involvement of additional maintainers and other expenses, I think that at this moment about 30 thousand dollars a month could be enough. More money — better product and faster development. A couple times less — it makes sense to resume the work on `core-js` full-time alone — sure, not as productive as it could be with a team.

- **I may be hired by a company where I will be able to work on Open-Source and Web standards**

  and that will give me the resources required for continuation of the work.

- **`core-js` will become a commercial project** if it will not receive an appropriate support from users

  It's problematic to create a commercial infrastructure around the current `core-js` packages, so most likely the new `core-js` major release will change the license. The free version will be significantly limited. All extra functionality will be paid for. `core-js` will continue to evolve appropriately and, in the scope of this project, many new tools will be created to ensure web compatibility. Sure, it will significantly reduce the spread of `core-js` and will cause problems for many developers, however, even some paying customers could be enough and my family will have money to pay the bills.

- A **slow death** in case I'll see that `core-js` is not required

  I have many ideas for commercial projects, I have a lot of good job offers — all this takes time, which now goes into `core-js` maintenance. It does not mean that I'll immediately completely stop maintaining `core-js`, I'll just maintain pro-rata donations. If they are at the current level, it will be only a few hours of maintenance a month instead of hundreds like now. The project will stop the upgrowth — maybe minor bugs will be fixed and compatibility data will be updated — this time is not enough for more. After a while, `core-js` will become just useless and will die.

I still hope for the first outcome since `core-js` is one of the key components of the modern digital infrastructure, but, looking at the present and the past, I am mentally getting ready for other options.

I will answer some angry comments in advance that I see regularly and that will definitely come up after this post:

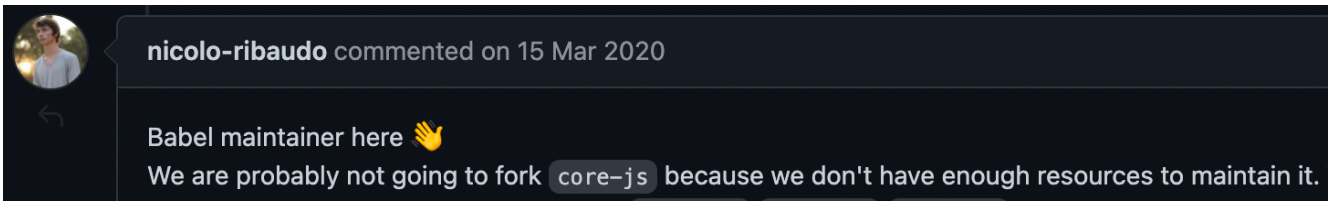
- **"Not a problem, we will just pin the `core-js` dependency."**

  Unlike most projects, `core-js` should be on the bleeding edge since `core-js` allows you to be on the bleeding edge of JavaScript: use the most recent JavaScript features and don't think about engines compatibility and bugs. However, the library has a good safety margin for the future. Maybe for a year or a couple, you will not have serious problems. After that, they will appear — polyfills will become obsolete, but still will be present in your bundles and will become just a useless ballast. You will not be able to use new features of the language and will face new bugs in JS engines.

- **"It's open-source, we will fork it, fuck off."**

  I see such comments regularly, someone even tries to scare me with a fork. I've said already too many times that **if someone will fork and properly maintain `core-js`, I'd be happy** — it makes no sense just to fork it without maintenance. Now I don't see anyone at all who tries to add something significant to `core-js` or at least contribute regularly. The project ought to follow up on each new JavaScript

engine release to update compatibility data, fix or at least take into account each new (no matter how significant) bug from each engine, take a look and implement each new JavaScript feature possible, do it maximally properly, test and take into account the specifics of each version of each modern or legacy engine. It's a hard work, are you ready and have the required knowledge and time for that? For example, when I was in prison, Babel said that they are not:



- **"We don't need `core-js`, many alternative projects are available."**

  Nobody is holding you. But where are those alternatives in real life? Sure, `core-js` is not the only polyfill of the JavaScript standard library. But all other projects are tens of times less popular than `core-js`, and it's not unreasonable — all of them provide only a small part of `core-js` functionality, they are not proper and complex enough, the number of cases where they can be used is significantly limited, they can't be properly integrated into your project in such a simple way and have other significant problems. In the case if proper alternatives existed, I would have stopped working on `core-js` a long time ago.

- **"We can drop IE support, so we no longer need polyfills."**

  As I wrote a just above, nobody is holding you. In some cases, polyfills are really not required and you can avoid them, but it's only a small part of all cases — almost the same as it was in the IE era. Of course, if you don't need IE support, polyfills will not expand your possibilities as much as it was with adding ES6 support to IE8. But even the most modern engines do not implement the most modern JavaScript features. Even the most modern engines contain bugs. Are you pretty sure that you and your team perfectly know all limitations of all engines that you support and can work around them? Even I sometimes may forget some quirks and missing features.

- **"You are an asshole, we will expel you from the FOSS community."**

  Yes, you're right. I'm such an asshole that gives you a chance to use modern JavaScript features in the real life, have been solving your cross-engine compatibility issues for many years, and had sacrificed for this more than anyone else. I'm such an asshole that just wants his son to be well-fed, wants his family to have enough money to pay the bills, so they don't need anything. Some options above suppose my departure from FOSS in favor of commercial software, so we'll see.

Now let's move away from the negative to the second half of this post where we will talk about things that would be nice to implement in `core-js` and the problems of polyfilling in general.

Roadmap

JavaScript, browsers, and web development are evolving at an amazing speed. The time when almost all of the `core-js` modules were required for all browsers is gone. The latest browsers have good standards support and, in the common use cases, they need only some percentage of the `core-js` modules for the most recent language features and bug fixes. Some companies are already dropping support for IE11 which was recently "buried" once more. However, even without IE, old browsers will always be there, bugs will happen in modern browsers too, and new language features will appear regularly and they will appear in

browsers with a delay anyway; so, if we want to use modern JS in development and minimize possible problems, polyfills stay with us for a long time, but they should continue to evolve.

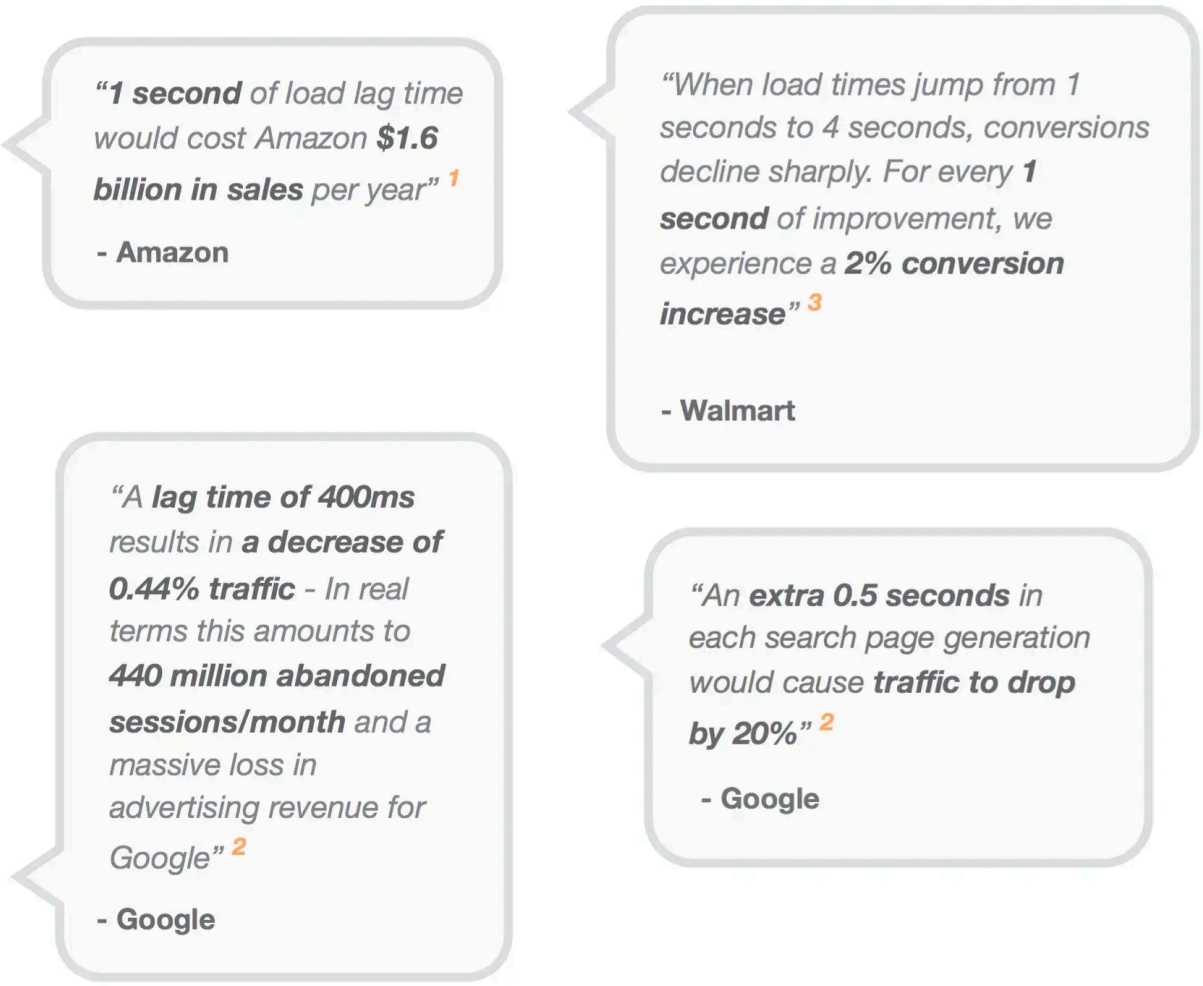
Here I will write (almost) nothing about adding new or improving existing specific polyfills (but, sure, it's one of the main parts of `core-js` development), let's talk about some other crucial moments without focusing on minor things. If it is decided to make a commercial project from `core-js`, the roadmap will be adapted to this outcome.

I am trying to keep `core-js` as compact as possible, but one of the main conceptions that it should follow is to be maximally useful in the modern web — the client should not load any unnecessary polyfills and polyfills should be maximally compact and optimized. Currently, a maximal `core-js` bundle size with early-stage proposals is about 220KB minified, 70KB gzipped — it's not a tiny package, it's big enough — it's like jQuery, LoDash, and Axios together — the reason is that the package covers almost the entire standard library of the language. The individual weight of each component is several times less than the weight of quite correct alternatives. It's possible to load only the `core-js` features that you use and in minimal cases, the bundle size can be reduced to some kilobytes. When `core-js` is used correctly, this is usually a couple of tens of kilobytes — however, there is something to strive for. Most pages contain pictures larger than the entire `core-js` bundle, most users have Internet speed in dozens of Mbps, so why is this concept so significant?

I don't want to repeat old posts about the cost of JavaScript in detail where you can read why adding JS increases the time when the user can start interacting with the page much more than adding a similar size picture — it's not only downloading, it's also parsing, compiling, evaluating the script, it blocks the page rendering.

In too many places the mobile Internet is not perfect and is still 3G or even 2G. In the case of 3G, the download of one full copy of `core-js` can take a couple of seconds. However, pages contain more than one copy of `core-js` and many other duplicated polyfills too often. Some (mainly mobile) Internet providers have very limited "unlimited" data plans and after a few gigabytes reduce the speed to a few Kbps. The connection speed is often limited for many other reasons too.

The speed of the page load equals revenue.



> Illustration is from a random post by googling

The size of `core-js` is constantly growing because of the addition of new or improvements to the existing polyfills. This issue also is a blocker for some big polyfills — the addition of `Intl`, `Temporal`, and some other features to `core-js` could increase the maximal bundle size by a dozen times up to a few megabytes.

One of the main `core-js` killer features is that it can be optimized with the usage of Babel, SWC, or manually, however, current approaches solve only a part of the problem. To properly solve them, the modern web requires a new generation of the toolkit that could be simply integrated into the current development stack. And in some cases, as you will see below, this toolkit could help to make the size of your web pages even less than just without `core-js`.

I already wrote about some of this in `core-js@3`, **Babel and a look into the future** post, but those were just raw ideas. Now they're in the stage of experimentation or even implementation.

Since the future of the project is uncertain, it makes no sense to write any specific dates here, I do not promise that all of this will be done shortly, but this is what should be strived for.

New major version

`core-js@3` was released about 4 years ago — it's been a long time. It's not a big problem for me to add some breaking changes (rather ensuring backward compatibility is often a challenge) and to mark a new version as a major release — it's a big problem for the users.

At this moment, about 25% of `core-js` downloads are critically obsolete `core-js@2`. Many users wanna update it to `core-js@3`, but because their dependencies use `core-js@2` they still use the obsolete version to avoid multiple copies (I saw such issues on GitHub in too many projects). Too frequent major updates would worsen such cases even more.

However, it's better not to get too obsessed with compatibility with older versions. The library contains too much that's not removed only for compatibility reasons. The absence of some long-needed breaking changes for someone will negatively affect the future. Judging by how the standards, the ecosystem, and the Web change, and how legacy accumulates, it's better to release a new major version each 2-3 years.

The addition of all the new things that we would like to see in the new major version would take many years, which is unacceptable. However, `core-js` follows SemVer and it makes sense to release a new major release at first with breaking changes (some of them below), most of the new features can be added in minor releases. In this case, such a release can take just about 2-3 months of full-time work and it can be the first `core-js` version that reduced the size compared to the previous -)

`core-js` package directly

Drop critically obsolete engines support

IE is dead. However, not for all — for many different reasons, someone is still forced to make or maintain websites that should work in IE. `core-js` is one of the main tools that makes life easier for them.

At this moment, `core-js` tries to support all possible engines and platforms, even ES3 — IE8-. But only a small part of developers using `core-js` needs support of ES3 engines — at this moment, the IE8- segment of browsers is about 0.1%. For many other users, it causes problems — bigger bundle size and slower runtime execution.

The main problem comes from supporting ES3 engines: most modern ES features are based on ES5 features, which aren't available in those old engines. Some features (like getters / setters) can't be polyfilled, so some polyfills (like typed arrays) can't work in IE8- at all. Some others require heavy workarounds. In cases where you need to polyfill only some simple features, the main part of the `core-js` size in the bundle is the implementation of ES5 methods (in the case of polyfilling a lot of features, it's only some percent, so this problem is related mainly to minimalistic bundles).

Even the simple replacement of internal fallbacks of ES5 features to implementations to direct usage of those native features reduces minimalistic `core-js` bundle size by 2+ times. After reworking the architecture, it will be reduced even more.

The IE9-10 segment of browsers already is also small — at this moment, the same 0.1%. But it makes no sense to consider dropping their support without dropping support of some other obsolete engines with similar or even greater restrictions, for example, Android 4.4.4 — in total, it's about 1%. Raising the lower bar higher than ES5 is a more difficult decision at least because of some non-browser engines. However, even dropping IE11 support in the future will not give as many benefits as dropping IE8- support would now.

ECMAScript modules and modern syntax

At this moment, `core-js` uses CommonJS modules. For a long time, it was the most popular JavaScript modules format, but now ECMAScript provides its own modules format and it's already very popular and supported *almost* everywhere. For example, Deno, like browsers, doesn't support CommonJS, but supports ES modules. `core-js` should get an ECMAScript modules version in the near future. But, for example, on NodeJS, ECMAScript modules are supported only in the modern versions — but on NodeJS `core-js` should work without transpiling / bundling even in ancient versions, Electron still does not support it, etc., so it's problematic to get rid of the CommonJS version immediately.

The situation with the rest of modern syntax is not so obvious. At this moment, `core-js` uses ES3 syntax. Initially, it was for maximal optimization since it should be pre-transpiled to old syntax anyway. But it was true only initially. Now, `core-js` just can't be properly transpiled in userland and should be ignored in transpiler configs. Why? Let's take a look, for example, at Babel transforms:

- A big part of transforms rely on modern built-ins, for example, transforms which use `@@iterator` protocol — yet `Symbol.iterator`, iterators, and all other related built-ins are implemented in `core-js` and absent before `core-js` loading.
- Another problem is transpiling `core-js` with transforms that inject `core-js` polyfills. Obviously, we can't inject polyfills into the place where they are implemented since it is circular dependencies.
- Some other transforms applied on `core-js` just break its internals — for example, the `typeof` transform (that should help with support of polyfilled symbols) breaks the `Symbol` polyfill.

However, the usage of modern syntax in polyfills code could significantly improve the readability of the source code, reduce the size and in some cases improve performance if polyfill is bundled for a modern engine, so it's time to think about rewriting `core-js` to modern syntax, making it transpilable by getting around those problems and publishing versions with different syntax for different use cases.

Web standards polyfills

I've been thinking about adding the most possible web standards (not only ECMAScript and closely related features) support to `core-js` for a long time. First of all, about the remaining features from the Minimum Common Web Platform API (what is it?), but not only about them. It could be good to have one bulletproof polyfills project for all possible web development cases, not only for ECMAScript. At the moment, the situation with the support of web standards in browsers is much worse than with the support of modern ECMAScript features.

One of the barriers preventing the addition of web standards polyfills to `core-js` was a significant increase of bundles' size, but I think that with current techniques of loading only the required polyfills and techniques which you can see below, we could add polyfills of web standards to `core-js`.

But the main problem is that it should not be naive polyfills. As I wrote above, today the correctness of ECMAScript features is not in a very bad shape almost universally, but we can't say this about web platform features. For example, a `structuredClone` polyfill was relatively recently added. When working on it, taking into account the dependencies, I faced **hundreds** of different JavaScript engines bugs — I don't remember when I saw something like that when I added new ECMAScript features — for this reason, the work on this simple method, that naively could be implemented within a couple hours, including resolving all issues and adding required features, lasted for several months. In the case of polyfills, better to do nothing than to do bad. The proper testing, polyfilling, and ensuring cross-platform compatibility web platform features require even more significant resources than what I spend on ECMAScript polyfills. So adding the maximum possible web standards support to `core-js` will be started only in case if I have such resources.

New approaches to tooling are more interesting

Someone will ask why it's here. What do tools, like transpilers, have to do with the `core-js` project? `core-js` is just a polyfill, and those tools are written and maintained by other people. Once I also thought that it is enough to write a great project with a good API, explain its possibilities, and when it becomes popular, it will acquire an ecosystem with proper third-party tools. However, over the years, I realized that this will not happen if you do not do, or at least not control, it yourself.

For example, for many years, instance methods were not able to be polyfilled through Babel `runtime`, but I explained how to do it too many times. Polyfilling via `preset-env` could not be used in real-life projects because of incomplete detection of required polyfills and a bad source of compatibility data, which I explained from the beginning. Because of such problems, I was forced to almost completely rewrite those tools in 2018-2019, for the `core-js@3` release, after that we got the current state of statically analysis-based tools for polyfills injecting.

I am sure that if the approaches below are not implemented in the scope of `core-js`, they will not be properly implemented at all.

To avoid some questions related to the following text: `core-js` tools will be moved to scoped packages — tools like `core-js-builder` and `core-js-compat` will become `@core-js/builder` and `@core-js/compat` respectively.

Not only Babel: plugins for transpilers and module bundlers

At this moment, some users are forced to use Babel only due to the need to automatically inject / optimize required polyfills. At this moment, Babel's `preset-env` and `runtime` are the only good enough and well-known ways to optimize usage of `core-js` with statical analysis. Historically, it happened because I helped Babel with polyfills. It does not mean that it's the only or the best place where it could be done.

Babel is only one of many transpilers. TypeScript is another popular option. Other transpilers are gaining popularity now, for example, SWC (that already contains a tool for automatic polyfilling / `core-js` optimization, but it's still not perfect). However, why do we talk about the transpilers layer? The bundlers layer and tools like `webpack` or `esbuild` (that also contains an integrated transpiler) are more interesting for the optimization of polyfills. Rome has been in development for several years and still is not ready, but its concept looks very promising.

One of the main problems with statical analysis-based automatic polyfilling on the transpiler layer is that usually not all files from the bundle are transpiled — for example, dependencies. If some of your dependencies need a polyfill of a modern built-in feature, but you don't use this built-in in your userland code, this polyfill will not be added to the bundle. Unnecessary polyfills import also will not be removed from your dependencies (see below). Moving automatic polyfilling to the bundlers layer fixes this problem.

Sure, writing or using such plugins in many places is difficult compared to Babel. For example, now without some extra tools you can't use plugins for custom transforms in TypeScript. However, where there's a will there's a way.

Automatic polyfilling / optimization of `core-js` should be available not only in Babel. It's almost impossible to write and maintain plugins for all transpilers and bundlers in the scope of the `core-js` project, but it's possible to do those things:

- Improve data provided by `core-js` (`@core-js/compat`) and tools for integration with third-party projects, they should be comprehensive. For example, "built-in definitions" are still on Babel's side that causing problems with their reuse in other projects.
- Since some tools already provide `core-js` integration, it makes sense to help them too, not just Babel.
- It makes sense to write and maintain plugins for some significant tools in the scope of the `core-js` project. Which? We will see.

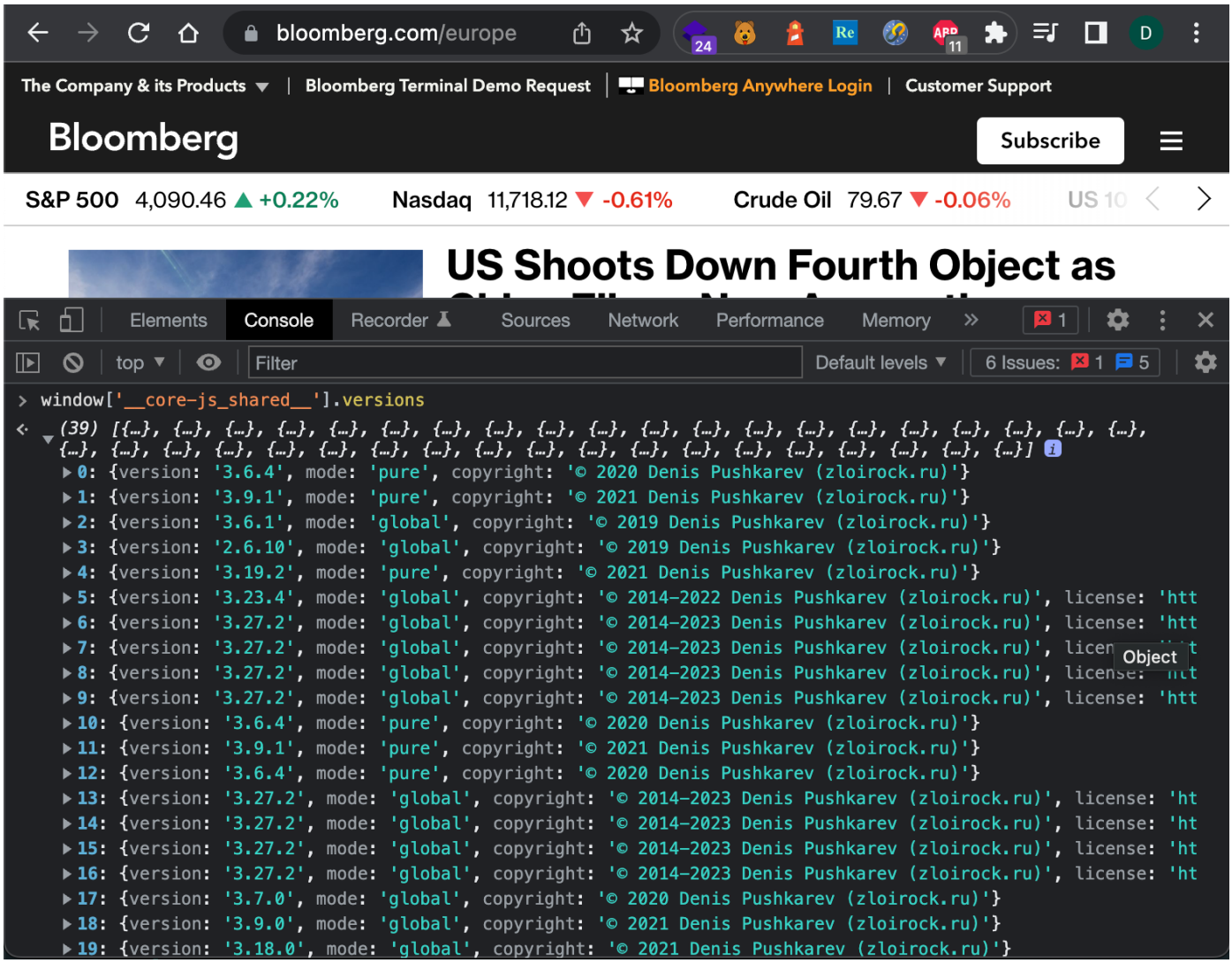
Polyfills collector

One of the problems of the statical analysis-based automatic polyfilling on the files layer (`usage` polyfilling mode of Babel `preset-env`) was explained above, but it's not the only problem. Let's talk about some others.

Your dependencies could have their own `core-js` dependencies and they can be incompatible with the `core-js` version that you use at the root of your project, so injecting `core-js` imports to your dependencies directly could cause breakage.

Projects often contain multiple entry points, multiple bundles, and, in some cases, the proper moving of all `core-js` modules to one chunk can be problematic and it could cause duplication of `core-js` in each bundle.

I already posted the `core-js` usage statistics above. In many cases, you could see the duplication of `core-js` — and it's only on the first loaded page of the application. Sometimes it's even like what we see on the Bloomberg website:



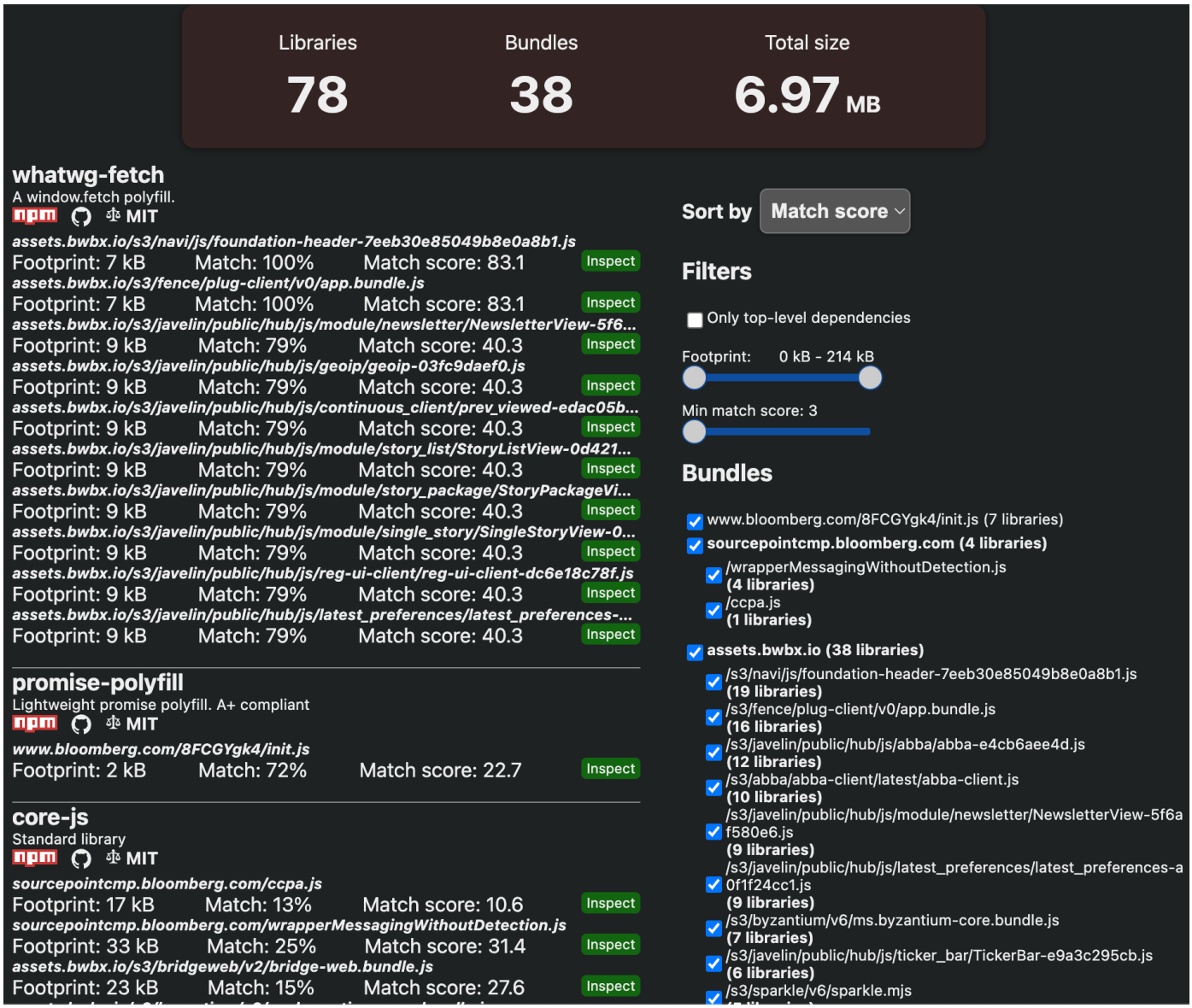
Some time ago this number was even higher. Of course, such a number of copies and various versions of `core-js` is not something typical, but a situation with several copies of `core-js` is too common as you saw above, affecting about half the websites with `core-js`. To prevent this **a new solution is required to collect all polyfills from all entry points, bundles and dependencies of the project in one place.**

Let's call a tool for this `@core-js/collector`. This tool should take an entry point or a list of entry points and should use the same statical analysis that's used in `preset-env`, however, this tool should not transform code or inject anything, should check full dependencies trees and should return a full list of required `core-js` modules. As a requirement, it should be simple to integrate into the current development stack. One possible way can be a new polyfilling mode in plugins, let's call it `collected` — that will allow loading all collected polyfills of the application in one place and remove the unnecessary (see below).

Removing unnecessary third-party polyfills

Now it's typical to see, for example, a dozen copies of `Promise` polyfills with the same functionality on a website — you load only one `Promise` polyfill from `core-js`, but some of your dependencies load `Promise` polyfills by themself — `Promise` polyfill from one more `core-js` copy, `es6-promise`, `promise-polyfill`, `es6-promise-polyfill`, `native-promise-only`, etc. But it's just ES6 `Promise` which is already completely covered by `core-js` — and available in most browsers without polyfills. Sometimes, due to this, the size of all polyfills in the bundle swells to several megabytes.

It's not an ideal illustration for this issue, many other examples would have been better, but since above we started to talk about the Bloomberg website, let's take a look at this site one more time. We have no access to the source code, however, we have, for example, such an awesome tool as `bundlescanner.com` (I hope that the Bloomberg team will fix it ASAP, so the result could be outdated).



As shown in the practice, since such analysis it's not a simple work, this tool detects only about half of libraries' code. However, in addition to 450 kilobytes of `core-js`, we see hundreds of kilobytes of other polyfills — many copies of `es6-promise`, `promise-polyfill`, `whatwg-fetch` (for the above reason, `core-js` *still* does not polyfill it), `string.prototype.codepointat`, `object-assign` (it's a *ponyfill* and the next section is about them), `array-find-index`, etc.

But how many polyfills were not detected? What's the size of all polyfills that this website loads? It seems a couple of megabytes. However, even for *very* old browsers, at most a hundred kilobytes are more than be enough... And this situation is not something unique — it's a too common problem.

Since many of those polyfills contain just a subset of `core-js` functionality, in the scope of `@core-js/compat`, we could collect data that will show if a module is an unnecessary third-party polyfill or not and, if this functionality is contained in `core-js`, a transpiler or bundler plugin will remove the import of this module or will replace it to the import of suitable `core-js` modules.

The same approach could be applied to get rid of dependencies from old `core-js` versions.

Globalization of pure version polyfills / ponyfills

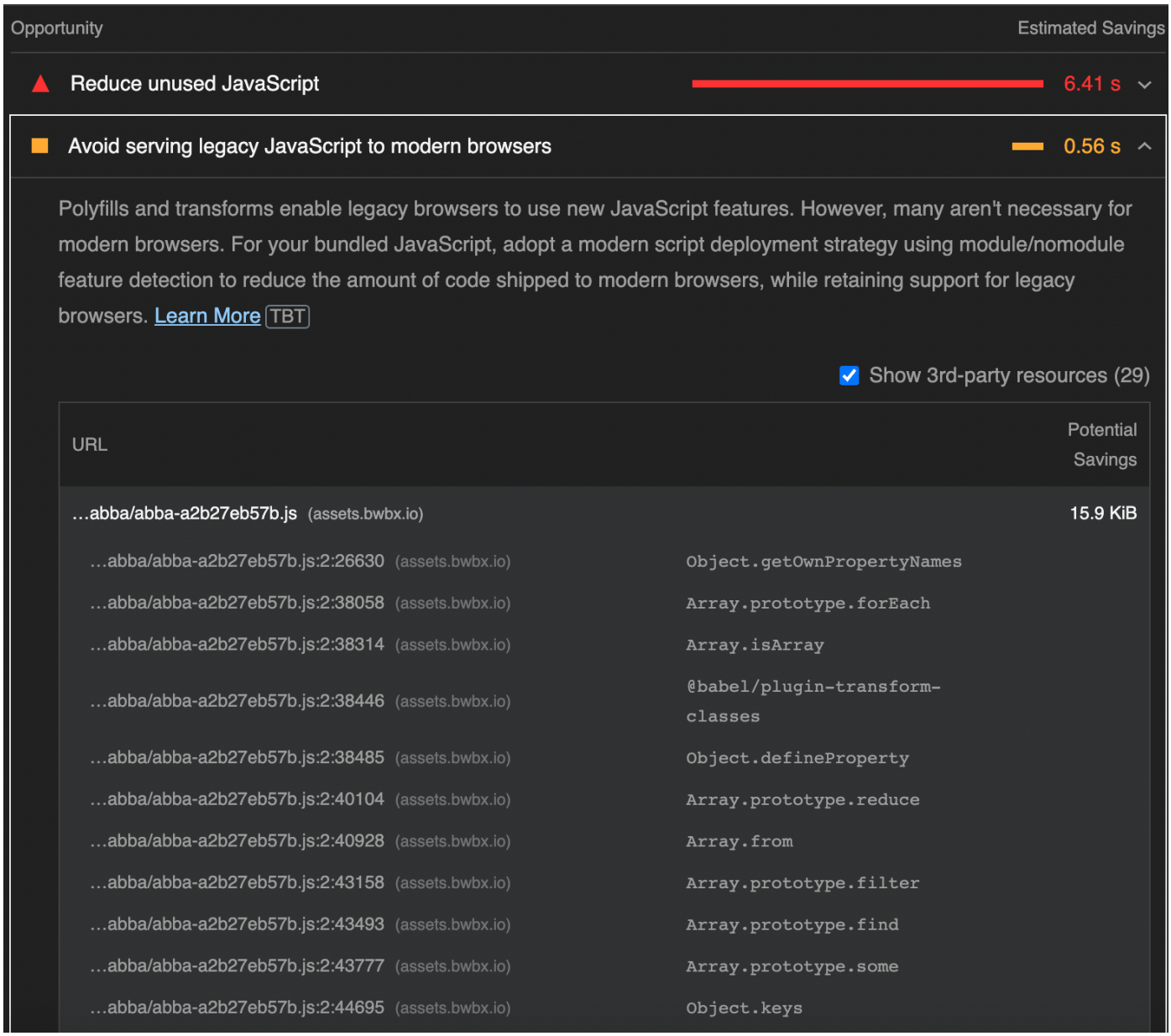
One more popular and similar issue is a duplication of polyfills from global and pure `core-js` versions. The pure version of `core-js` / `babel-runtime` is intended for usage in libraries' code, so it's a normal situation if you use a global version of `core-js` and your dependencies also load some copies of `core-js` without global namespace pollution. They use different internals and it's problematic to share similar code between them.

I'm thinking about resolving this issue on the transpiler or bundler plugins side similarly to the previous one (but, sure, a little more complex) — we could replace imports from the pure version with imports from the global version and remove polyfills unnecessary for the target engines.

That also could be applied to third-party ponyfills or obsolete libraries that implement something already available in the JS standard library. For example, the usage of `has` package can be replaced by `Object.hasOwn`, `left-pad` by `String.prototype.padStart`, some `lodash` methods by related modern built-in JS methods, etc.

Service

Loading the same polyfills, for example, in IE11, iOS Safari 14.8, and the latest Firefox is wrong — too much dead code will be loaded in modern browsers. At this moment, a popular pattern is the use of 2 bundles — for "modern" browsers that will be loaded if native modules are supported, `<script type="module">`, and for obsolete browsers which do not support native modules, `<script nomodule>` (a little harder in a practice). For example, Lighthouse can detect some cases of polyfills that are not required with the `esmodules` target, let's check the long-suffering Bloomberg website:



Lighthouse shows just about 200KB in all resources, 0.56s. Let's remember that the site contains about a couple of megabytes of polyfills. Now Lighthouse detects less than half of the features that it should, but even with another half, it's only a little part of all loaded polyfills. Where are the rest? Are they really required for a modern browser? The problem is that the lower bar of native modules support is too low — "modern" browsers will, in this case, need most of the polyfills of stable JS features that are required for old IE, so a part of polyfills is shown in the "unused JavaScript" section that takes 6.41s, a part is not shown at all...

From the very beginning of work on `core-js`, I've been thinking about creating a web service that serves only the polyfills needed for the requesting browser.

The availability of a such service is the only aspect in which `core-js` have lagged behind another project. `polyfill-service` from Financial Times is based on this conception and it's a great service. The main problem with this project — it's a great service that uses poor polyfills. This project polyfills only a little part of the ECMAScript features that `core-js` provides, most of the polyfills are third-party and are not designed to work together, too many don't properly follow specs, too unpolished or just dangerous for usage (for example, `WeakMap` looks like a step-by-step implementation of the spec text, but the absence of some non-spec magic cause memory leaking and linear access time that makes it harmful, but here's more — instead

of patching, fixing and reusing of native implementation in engines like IE11 where it's available, but does not accept an iterable argument, `WeakMap` will be completely replaced). Some good developers try to fix this from time to time, but polyfills themselves are given unforgivably little time, so it's still too far from something that could be recommended for usage.

Creating such a service in the proper form requires the creation and maintenance of many new components. I work on `core-js` alone, the project does not have the necessary support from any company, and the development is carried out with pure enthusiasm, I need to look for funds to feed myself and my family, so I have no time and other resources required for that. However, in the scope of other tasks, I already made some required components, and discussions with some users convinced me that creating a maximally simplified service that you could start on your own server could be enough.

We already have the best set of polyfills, the proper compatibility data, and the builder which could already create a bundle for a target browser. The previously mentioned `@core-js/collector` could be used for optimization — getting only the required subset of modules, plugins for transpilers / bundlers — for removing unnecessary polyfills. Missing a tool for the normalization of the user agent and a service that will bind those components together. Let's call it `@core-js/service`.

What should it look like in a perfect world?

- You bundle your project. A plugin on the bundler's side removes all polyfill imports (including third-party, without global pollution, from your dependencies, etc.). Your bundles will not contain any polyfills.
- You run `@core-js/service`. When you run it, `@core-js/collector` checks all your frontend codebase, all your entry points, including dependencies, and collects a list of all required polyfills.
- A user loads a page and requests a polyfill bundle from the service. The service gives the client a bundle compiled for the target browser that contains the required subset of polyfills and uses allowed syntax.

So, with this complex of tools, modern browsers will not load polyfills at all if they are not required, old browsers will load only the required and maximally optimized polyfills.

Most of the above is about minimizing the size of polyfills sent to the client — but these are just a little subset of the concepts that it would be good to implement in the scope of `core-js`, however, I think that it's enough to understand that still requires a huge work and this work could significantly improve web development. Whether it will be implemented in practice and whether it will be available as FOSS or as a commercial project is up to you.

Conclusion

This was the last attempt to keep `core-js` as a free open-source project with a proper quality and functionality level. It was the last attempt to convey that there are real people on the other side of open-source with families to feed and problems to solve.

If you or your company use `core-js` in one way or another and are interested in the quality of your supply chain, support the project:

- **Open Collective**

- Patreon
- Boosty
- Bitcoin (bc1qlea7544qtsmj2rayg0lthvza9fau63ux0fstcz)
- Alipay

Contact me if you can offer a good job on Web-standards and open-source.

Feel free to add comments to this post here.

Denis Pushkarev, February 14th 2023